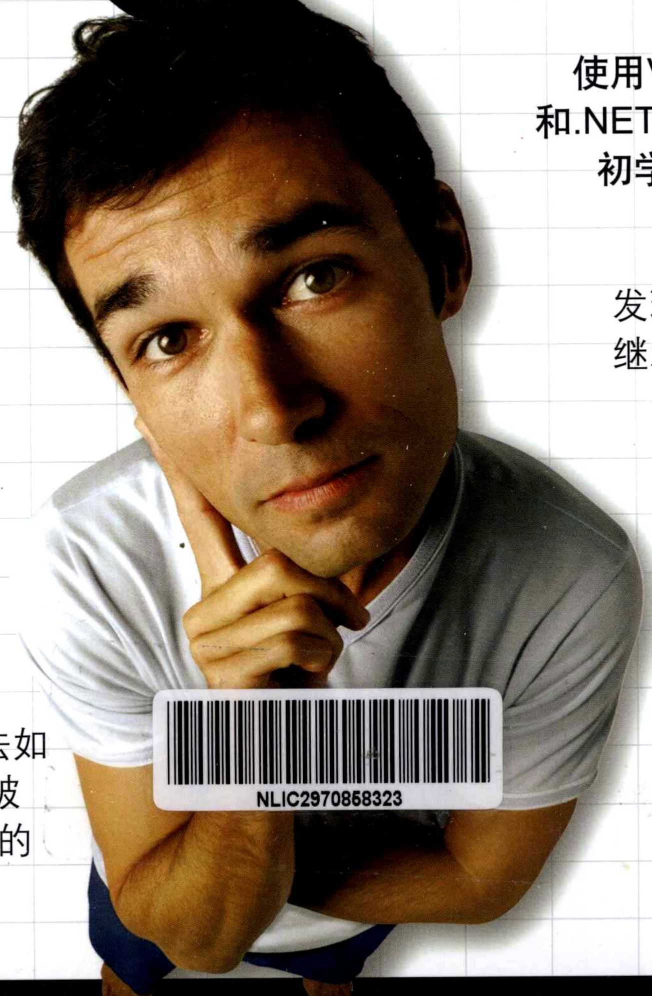
重视大脑的学习指南

第二版
涵盖C# & .Net 4.0
和Visual Studio 2010

# Head First C#（中文版）

使用Visual C#
和.NET实际编程的
初学者指南



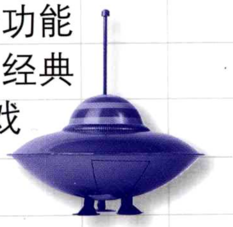


用LINQ控制
你的数据



发现抽象和
继承的秘密

构建一个功能
完备的经典
怀旧游戏





了解扩展方法如
何帮助Sue打破
Objectville公司的
规定



了解Jim如何使用
泛型集合处理
他的数据

NLIC2970858323

Andrew Stellman & Jennifer Greene 著
林琪 刘晓兵 等译

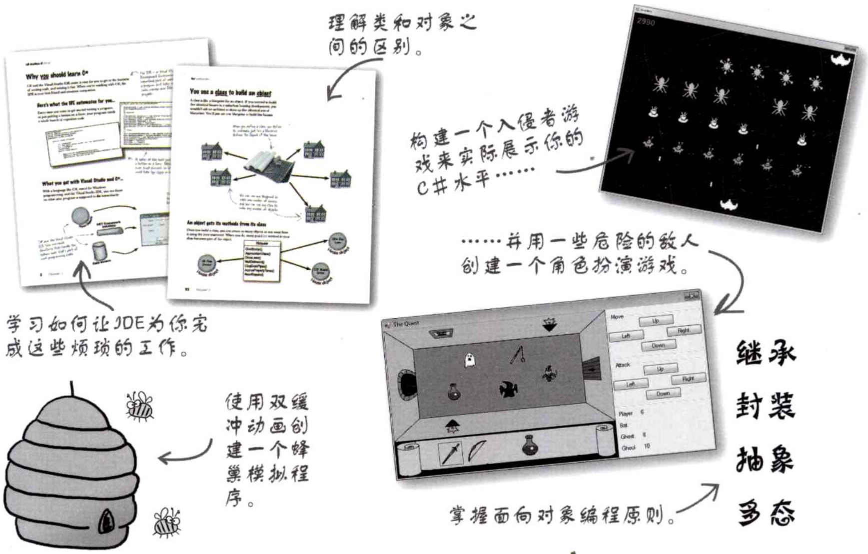
O'REILLY® 中国电力出版社

# Head First C#（中文版）

Programming Languages/Microsoft C#/.NET

## 从这本书能学到什么？

《Head First C#（第二版）》是学习C#编程、.NET Framework和Visual Studio IDE的绝佳途径。这本书是针对你的大脑特别制作的，涵盖C# & .NET 4.0和Visual Studio 2010，讲授了从继承到串行化等。你可以利用LINQ查询数据，可以绘制图形和完成动画，还将了解关于类和面向对象编程的所有内容，这些知识将通过构建游戏、动手建立工程，以及解决问题来获得。你将成为一名高水平的C#程序员，而且会有一个轻松愉悦的学习过程！




## 为什么这本书如此与众不同？

我们认为，你的时间如此宝贵，不应过多地浪费在与新概念的斗争中。通过使用认知科学和学习理论的最新研究成果，你将享受一种多感官学习体验，本书采用了一种专门为你的大脑而设的丰富格式娓娓道来，而不是长篇累牍地说教，让你昏昏欲睡。

"如果你想深入地学习C#，尽享其中的快乐，不要迟疑，这正是你需要的书。"

——Andy Parker，C#编程新手

"《Head First C#（第二版）》可以指导各种各样的初学者切实掌握C#和.NET Framework，并从此长期使用这个工具开发出优秀的声品。"

——Chris Burrows，微软C#编译器团队开发人员

"《Head First C#（第二版）》是一本超级有趣的教程，让人过目难忘的例子和好玩的练习比比皆是。"

——Joseph Albahari，Egton Medical信息系统的C#设计建构师，这是英国最大的重要保健软件提供商，《C# 4.0 in a Nutshell》合作者


O'Reilly Media, Inc.授权中国电力出版社出版

O'REILLY®
oreilly.com.cn
headfirstlabs.com

ISBN 978-7-5123-3127-3

定价：128.00元

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

# Head First C#

**第二版**




藏书
NLIC
NLIC2970858323

Andrew Stellman,
Jennifer Greene 著
林琪 刘晓兵 等译

O'REILLY®

*Beijing • Cambridge • Köln • Sebastopol • Tokyo*
O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

**图书在版编目（CIP）数据**

Head First C#（第二版）/（美）施特尔曼（Stellman, A.），（美）格林（Greene, J.）著；林琪等译，－北京：中国电力出版社，2012.6

书名原文：Head First C#, Second Edition

ISBN 978-7-5123-3127-3

I. ①H… II.①施… ②格… ③林… III. ①C语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2012）第113725号

北京市版权局著作权合同登记

图字：01-2010-7920号

©2010 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2012. Authorized translation of the English edition, 2010 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc.出版2010。

简体中文版由中国电力出版社出版，2012。英文原版的翻译得到 O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/　Head First C#（第二版）
书　　号/　ISBN 978-7-5123-3127-3
责任编辑/　刘炽
封面设计/　Louise Barr，Karen Montgomery，张健
出版发行/　中国电力出版社
地　　址/　北京市东城区北京站西街19号（邮政编码100005）
印　　刷/　航远印刷有限公司
开　　本/　880毫米×1230毫米　20开本　42印张　1130千字
版　　次/　2013年1月第1版　2013年1月北京第1次印刷
印　　数/　0001－3000册
定　　价/　128.00元（册）

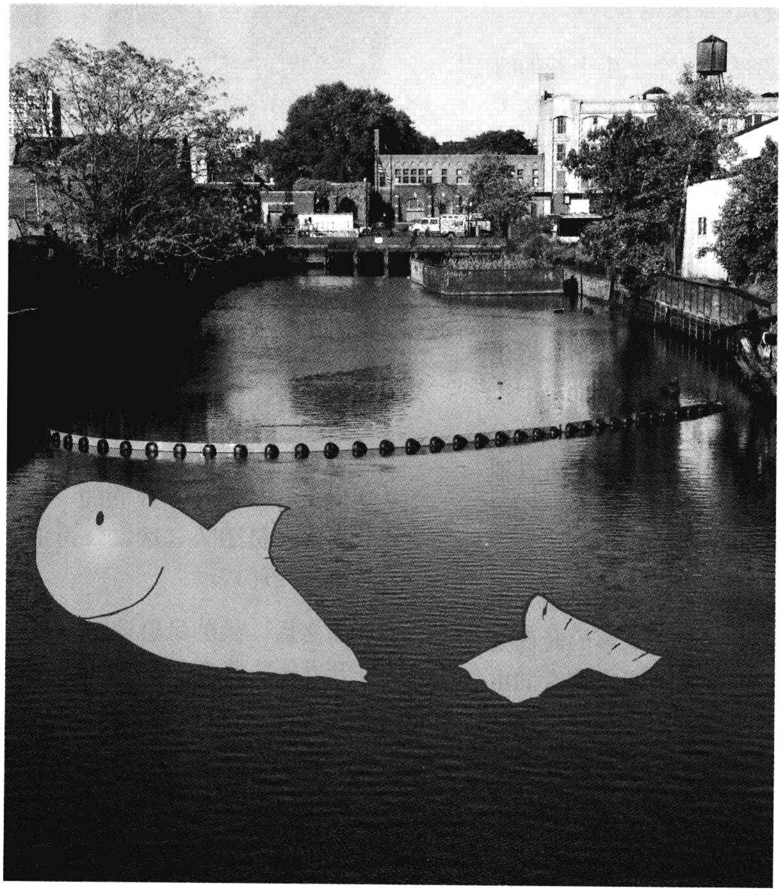
**敬 告 读 者**

本书封底贴有防伪标签，刮开涂层可查询真伪

本书如有印装质量问题，我社发行部负责退换

**版 权 专 有　　翻 印 必 究**

这本书仅献给2007年4月17日游入布鲁克林的灰鲸。



虽然你只在我们的运河里待了一天，
但你将永远留在我们的心里。




这张照片（还有Gowanus Canal的照片）由Nisha Sondhe拍摄

Andrew Stellman，虽然是一个土生土长的纽约人，却曾两次居住在匹兹堡。第一次是从卡耐基梅隆计算机科学学院毕业。第二次则是他和Jenny开始着手开展他们的咨询业务，并为O'Reilly公司写他们的第一本书。

搬回故乡后，他在大学毕业后的第一份工作是在百代唱片公司EMI-Capitol Records做一名程序员，这不无道理，因为他曾在LaGuardia音乐艺术和表演艺术学校学习大提琴和爵士乐吉它。他和Jenny的第一次共事就是在这家财务软件公司，在那里他管理着一个程序员团队，所以独享特权，可以与一些了不起的程序员共事多年，并很高兴地从他们那里学到不少东西。

平常不写书时，Andrew会忙着写一些没用（但有趣）的软件，玩音乐（不过，更多的时间是打电子游戏），做一些常常发出奇怪声音的电路试验，学中国的太极拳和日本的合气道。他有一个女朋友Lisa，还养着一只波美拉尼亚种小狗。

Jennifer Greene，在大学里学的是哲学，不过，与这个领域中的所有其他人一样，光凭哲学没办法找到工作。幸运的是，她是一位优秀的软件测试人员，最早在一个互联网服务公司从事这个工作，这也是她第一次切实感觉到项目管理的意义。

她于1998年搬到纽约，在一家财务软件公司做软件测试工作。她在新成立的一家很棒的公司管理着一个测试人员团队（这家公司主要研究人工智能和自然语言处理）。

自那之后，她的足迹遍布世界各地，曾与不同的软件开发团队共事，并且构建了很多相当酷的工程。

她喜欢旅游、看好莱坞电影、看漫画书，玩PS3游戏（特别是LittleBigPlanet），另外还有一只机灵的小狗陪伴左右。

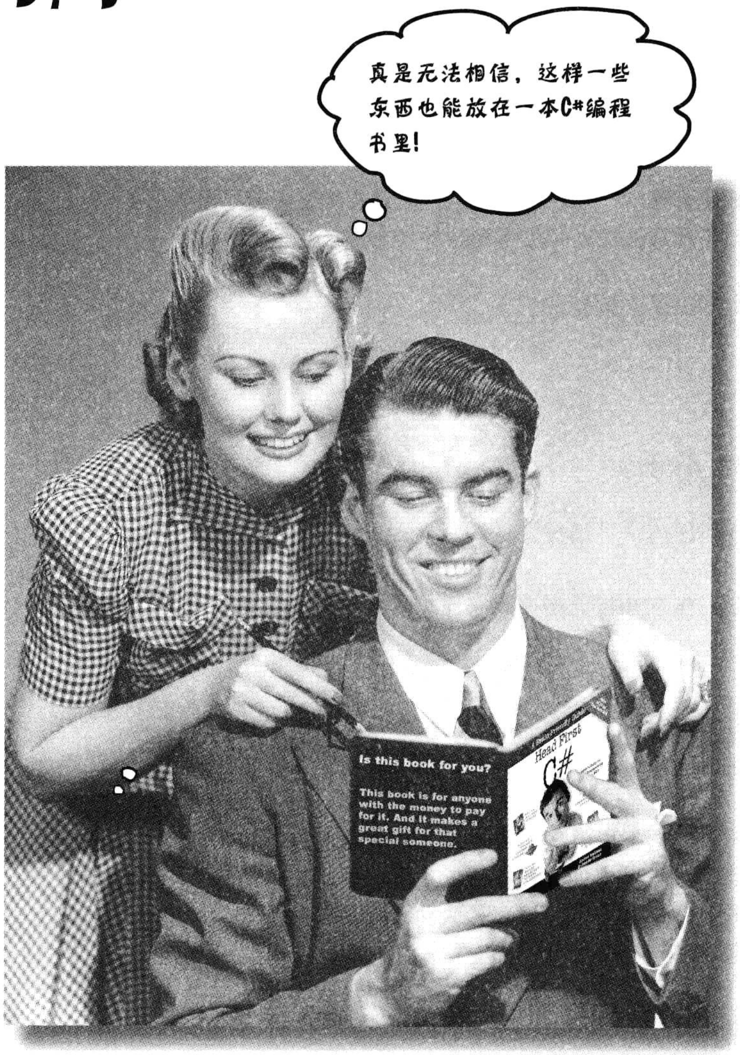
Jenny和Andrew从1998年第一次遇见以来一直都在开发软件，写有关软件工程的书。他们的第一本书《Applied Software Project Management》由O'Reilly在2005年出版。2007年出版了他们的第一本Head First系列图书《Head First PMP》。

他们在2003年创立了Stellman & Greene咨询公司，当时有科学家在研究越战老兵遭受除草剂危害的问题，他们为这些科学家建立一个很棒的软件项目。不编写软件或写书时，他们会参加软件工程师、架构师和项目经理的会议，并且频频发表演讲。

可以访问他们的博客“构建更棒软件”：http://www.stellman-greene.com

## 如何使用这本书

# 引子




有一个问题真是听得我们耳朵都磨出茧了，这就是："你们到底为什么要把这样一些东西放在一本C#编程书里呢？"这一章就来回答这个问题。

# 谁适合看这本书？

如果对下面的所有问题都能肯定地回答“是”：

1. 你想学C#吗？

2. 你喜欢修修补补吗？是不是喜欢亲自动手，在实践中学习，而不是只看不做，纸上谈兵？

3. 你是不是更喜欢一种轻松的氛围，就像在晚餐餐桌上交谈一样，而不愿意被动地听枯燥乏味的技术报告？

那么，这本书正是你需要的。

# 谁可能不适合看这本书？

如果满足下面任何一种情况：

1. 是不是一想到要写大量代码就让你头疼，甚至有些紧张？

2. 你本身是不是已经堪称一个很棒的C++或Java程序员，正在找一本参考书？

3. 你是不是对新鲜事物都畏首畏尾？只喜欢简单的样式，而不敢尝试把条纹和格子混在一起看看？你是不是觉得，如果把C#概念都拟人化了，这样的一本书肯定不是一本正经八百的技术书？



那么，这本书将不适合你。

[来自市场的声音：只要买得起，都可以拥有这本书。]

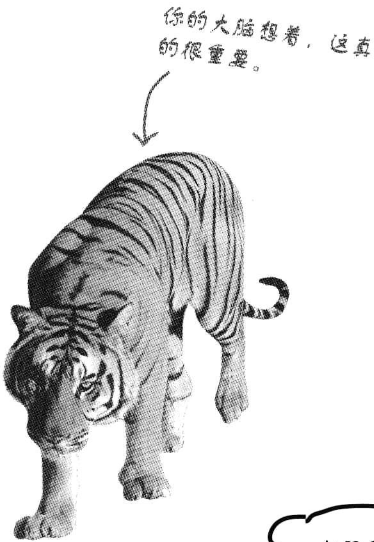
# 我们知道你在想什么。

“这算一本正式的C#编程书吗？”

“这些图用来做什么？”

“我真能这样学吗？”

# 我们也知道你的大脑正在想什么。




你的大脑总是渴求一些新奇的东西。它一直在搜寻、审视、期待着不寻常的事情发生。大脑的构造就是如此，正是这一点才让我们不至于墨守成规，能够与时俱进。

我们每天都会遇到许多按部就班的事情，这些事情很普通，对于这样一些例行的事情或者平常的东西，你的大脑又是怎么处理的呢？它的做法很简单，就是不让这些平常的东西妨碍大脑真正的工作。那么什么是大脑真正的工作呢？这就是记住那些确实重要的事情。它不会费心地去记乏味的东西，就好像大脑里有一个筛子，这个筛子会筛掉“显然不重要”的东西，如果遇到的事情枯燥乏味，这些东西就无法通过这个筛子。

那么你的大脑怎么知道到底哪些东西重要呢？打个比方，假如你某一天外出旅行，突然一只大老虎跳到你面前，此时此刻，你的大脑还有身体会做何反应？

神经元会“点火”，情绪爆发，释放出一些化学物质。

好了，这样你的大脑就会知道……

## 这肯定很重要！可不能忘记了！




不过，假如你正待在家里或者坐在图书馆里，这里很安全、很舒适，肯定没有老虎。你正在刻苦学习，准备应付考试。也可能想学一些比较难的技术，你的老板认为掌握这种技术需要一周时间，最多不超过十天。

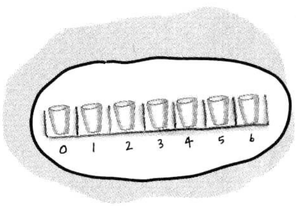
这就存在一个问题。你的大脑很想给你帮忙。它会努力地把这些显然不太重要的内容赶走，保证这些东西不去侵占本不算充足的脑力资源。这些资源最好还是用来记住那些确实重要的事情，比如大老虎，遭遇火灾险情等。再比如，你的大脑会让你记住，绝对不能把聚会时狂欢的照片放在你的Facebook网页上。

没有一种简单的办法来告诉大脑：“嘿，大脑，真是谢谢你了，不过不管这本书多没意思，也不管现在我对它多么无动于衷，但我确实希望你能把这些东西记下来。”

# 我们认为"Head First"读者就是要学习的人。

那么，怎么学习呢？首先必须获得知识，然后保证自己确实不会忘记。这可不是填鸭式的硬塞。根据认知科学、神经生物学和教育心理学的最新研究，学习的途径相当丰富，绝非只是通过书本上的文字。我们很清楚怎么让你的大脑兴奋起来。

**下面是一些Head First学习原则：**




**看得到**。与单纯的文字相比，图片更能让人记得住，通过图片，学习效率会更高（对于记忆和传递型的学习，甚至能有多达89%的效率提升）。而且图片更能让人看懂。以往总是把图片放在一页的最下面，甚至放在另外的一页上，与此不同，**把文字放在与之相关的图片内部，或者在图片的周围写上相关文字**，学习者的能力就能得到多至两倍的提高，从而能更好地解决有关的问题。

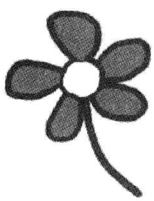


**采用一种针对个人的交谈式风格**。最新的研究表明，如果学习过程中采用一种第一人称的交谈方式直接向读者讲述有关内容，而不是用一种干巴巴的语调介绍，学生在学习之后的考试中成绩会提高40%。正确的做法是讲故事，而不是做报告。要用通俗的语言。另外不要太严肃。如果你面对着这样两个人，一个是你在餐会上结识的很有意思的朋友，另一个人学究气十足，喋喋不休地对你说教，在这两个人中，你会更注意哪一个呢？

**让学习的人想得更深**。换句话说，除非你很积极地让神经元活动起来，否则你的头脑里什么也不会发生。必须引起读者的好奇，促进、要求并鼓励读者去解决问题、得出结论、产生新的知识。为此，需要发出挑战，留下练习题和拓宽思路的问题，并要求读者完成一些实践活动，让左右脑都开动起来，而且要利用到多种思维。

**引起读者的注意，而且要让他一直保持注意**。我们可能都有过这样的体验，"我真的想把这个学会，不过看过一页后实在是让我昏昏欲睡"。你的大脑注意的是那些不一般、有意思、有些奇怪、抢眼的、意料之外的东西。学习一项有难度的新技术并不一定枯燥。如果学习过程不乏味，你的大脑很快就能学会。





**影响读者的情绪**。现在我们知道了，记忆能力很大程度上取决于所记的内容对我们的情绪有怎样的影响。如果是你关心的东西，就肯定记得住。如果让你感受到了什么，这些东西就会留在你的脑海中。不过，我们所说的可不是什么关于男孩与狗的伤心故事。这里所说的情绪是惊讶、好奇、觉得有趣、想知道"什么……"还有就是一种自豪感，如果你解决了一个难题，学会了所有人都觉得很难的东西，或者发现你了解的一些知识竟是那些自以为无所不能的傲慢家伙所不知道的，此时就会有一种自豪感油然而生。



# 元认知：有关思考的思考

如果你真的想学，而且想学得更快、更深，就应该注意你怎样才会专注起来，考虑自己是怎样思考的，并了解你的学习方法。

我们中间大多数人长这么大可能都没有上过有关元认知或学习理论的课程。我们想学习，但是很少有人教我们怎么来学习。

不过,这里可以做一个假设，如果你手上有这本书，你想学如何用C#构建程序，而且可能不想花太多时间。如果你想把这本书中读到的知识真正用起来，就需要记住你读到的所有内容。为此，必须理解这些内容。要想最大程度地利用这本书或其他任何一本书，或者掌握学习经验，就要让你的大脑负起责来，要求它记住这些内容。




怎么做到呢？技巧就在于要让你的大脑认为你学习的新东西确实很重要，对你的生活有很大影响。就像老虎出现在面前一样。如若不然，你将陷入旷日持久的拉锯战中，虽然你很想记住所学的新内容，但是你的大脑却会竭尽全力地把它们拒之门外。

### 那么究竟怎样才能让你的大脑把C#看做是一只饥饿的老虎呢？

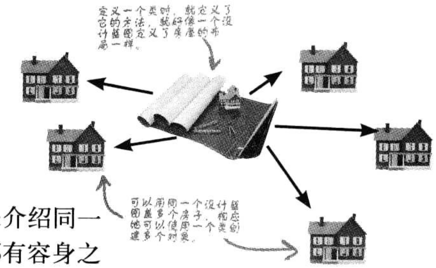
这有两条路，一条比较慢，很乏味。另一条路不仅更快，还更有效。慢方法就是大量地重复。你肯定知道，如果反反复复地看到同一个东西，即便再没有意思，你也能学会并记住。如果做了足够的重复，你的大脑就会说，“尽管看上去这对他来说好像不重要，不过，既然他这样一而再、再而三地看同一个东西，所以我觉得这应该是重要的。”

更快的方法是尽一切可能让大脑活动起来，特别是开动大脑来完成不同类型的活动。如何做到这一点呢？上一页列出的学习原则正是一些主要的可取做法，而且经证实，它们确实有助于让你的大脑全力以赴。例如，研究表明，把文字放在所描述图片的中间（而不是放在这一页的别处，比如作为标题，或者放在正文中），这样会让你的大脑更多地考虑这些文字与图片之间有什么关系，而这就会让更多的神经元点火。让更多的神经元点火 = 你的大脑更有可能认为这些内容值得关注，而且很可能需要记下来。

交谈式风格也很有帮助，当人们意识到自己在与“别人”交谈时，往往会更专心，这是因为他们总想跟上谈话的思路，并能做出适当的发言。让人惊奇的是，大脑并不关心“交谈”的对象究竟是谁，即使你只是与一本书“交谈”，它也不会在乎！另一方面，如果写作风格很正统、干巴巴的，你的大脑就会觉得，这就像坐在一群人当中被动地听人做报告一样，很没意思，所以不必在意对方说的是什么，甚至可以打瞌睡。

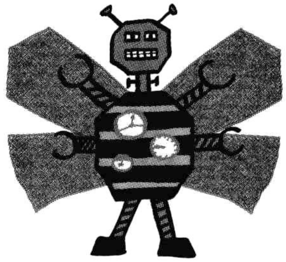
不过，图片和交谈风格还只是开始而已，能做的还有很多。

# 我们是这么做的：



我们用了很多图，因为你的大脑更能接受看得见的东西，而不是纯文字。对你的大脑来说，一幅图顶得上一千个字。如果既有文字又有图片，我们会把文字放在图片当中，因为文字处在所描述的图片中间时，大脑的工作效率更高，倘若把这些描述文字作为标题，或者“淹没”在别处的大段文字中，就达不到这种效果了。

我们采用了重复手法，会用不同方式，采用不同类型的媒体，运用多种思维手段来介绍同一个东西，目的是让有关内容更有可能储存在你的大脑中，而且在大脑中多个区域都有容身之地。



我们会用你想不到的方式运用概念和图片，因为你的大脑喜欢新鲜玩艺。在提供图和思想时，至少会含着一些情绪因素，因为如果能产生情绪反应，你的大脑就会投入更大的注意。而这会让你感觉到这些东西更有可能要被记住，其实这种感觉可能只是很点幽默，让人奇怪或者比较感兴趣而已。

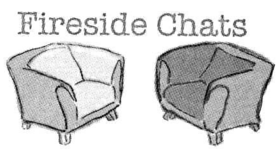
我们采用了一种针对个人的交谈式风格，因为当你的大脑认为你在参与一个会谈，而不是被动地听一场演示汇报时，它就会更加关注。即使你实际上在读一本书，也就是说在与书“交谈”，而不是真正与人交谈，但这对你的大脑来说并没有什么分别。

在这本书里，我们加入了80多个实践活动，因为与单纯的阅读相比，如果能实际做点什么，你的大脑会更乐于学习，更愿意去记。这些练习都是我们精心设计的，有一定的难度，但是确实能做出来，因为这是大多数人所希望的。




我们采用了多种学习模式，因为尽管你可能想循序渐进地学习，但是其他人可能希望先对整体有一个全面的认识，另外可能还有人只是想看一个例子。不过，不管你想怎么学，要是同样的内容能以多种方式来表述，这对每一个人都会有好处。

这里的内容不只是单单涉及左脑，也不只是让右脑有所动作，我们会让你的左右脑都开动起来，因为你的大脑参与得越多，你就越有可能学会并记住，而且能更长时间地保持注意力。如果只有一半大脑在工作，通常意味着另一半有机会休息，这样你就能更有效率地学习更长时间。

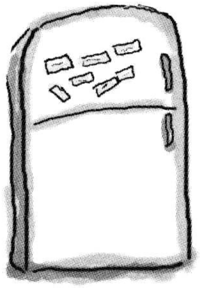



我们会讲故事，留练习，从多种不同的角度来看同一个问题，这是因为，如果要求大脑做一些评价和判断，它就能更深入地学习。

我们会给出一些练习，还会问一些问题，这些问题往往没有直截了当的答案，通过克服这些挑战，你就能学得更好，因为让大脑真正做点什么的话，它就更能学会并记住。想想吧，如果只是在体育馆里看着别人流汗，这对于保持你自己的体形肯定不会有什么帮助，正所谓临渊羡鱼，不如退而结网。不过另一方面，我们会竭尽所能不让你钻牛角尖，把劲用错了地方，而是能把工夫用在点子上。也就是说，你不会为搞定一个难懂的例子而耽搁，也不会花太多时间去弄明白一段艰涩难懂而且通篇行话的文字，我们的描述也不会太过简洁而让人无从下手。



我们用了拟人手法。在故事中，在例子中，还有在图中，你都会看到人的出现，这是因为你本身是一个人，不错，这就是原因。如果和人打交道，相对于某件东西而言，你的大脑会更为关注。



# 可以用下面的方法让你的大脑就范

好了，我们该做的已经做了，剩下的就要看你自己的了。以下提示可以作为一个起点：听一听你的大脑是怎么说的，弄清楚对你来说哪些做法可行，哪些做法不能奏效。要尝试新鲜事物。

把这一页撕下来，贴到你的冰箱上。

---

**1 慢一点。你理解的越多，需要记的就越少。**

不要光是看看就行了。停下来，好好想一想。书中提出问题的时候，你不要直接去翻答案。可以假想真的有人在问你这个问题。你让大脑想得越深入，就越有可能学会并记住它。

**2 做练习，自己记笔记。**

我们留了练习，但是如果这些练习的解答也由我们一手包办，那和有人替你参加考试有什么分别？不要只是坐在那里看着练习发呆。拿出笔来，写一写、画一画。大量研究都证实，学习过程中如果能实际动动手，这将改善你的学习。

**3 阅读“没有傻问题”部分。**

顾名思义，这些问题不是可有可无的旁注，它们绝对是核心内容的一部分！千万不要跳过去不看。

**4 上床睡觉之前不要再看别的书，至少不要看其他有难度的东西。**

学习中有一部分是在你合上书之后完成的（特别是，要把学到的知识长久地记住，这往往无法在看书的过程中做到）。你的大脑也需要有自己的时间，这样才能再做一些处理。如果在这段处理时间内你又往大脑里灌输了新的知识，那么你刚才学的一些东西就会丢掉。

**5 要喝水，而且要多喝点水。**

能提供充足的液体，你的大脑才能有最佳表现。如果缺水（可能在你感觉到口渴之前就已经缺水了），学习能力就会下降。

**6 讲出来，而且要大声讲出来。**

说话可以刺激大脑的另一部分。如果你想看懂什么，或者想更牢地记住它，就要大声地说出来。更好的办法是，大声地解释给别人听。这样你会学得更快，而且可能会有以前光看不说时不曾有的新发现。

**7 听听你的大脑怎么说。**

注意一下你的大脑是不是负荷太重了。如果发现自己开始浮光掠影地翻看，或者刚看的东西就忘记了，这说明你该休息一会了。达到某个临界点时，如果还是一味地向大脑里塞，这对于加快学习速度根本没有帮助，甚至还可能影响正常的学习进程。

**8 要有点感觉。**

你的大脑需要知道这是很重要的东西。要真正融入到书中的故事里。为书里的照片加上你自己的图题。你可能觉得一个笑话很憋脚，不太让人满意，但这总比根本无动于衷要好。

**9 编写大量软件!**

要学习编程，没有别的办法，只能通过编写大量代码。这本书正是要这么做。编写代码是一种技巧，要想在这方面擅长，只能通过实践。我们会给你提供大量实践的机会：每一章都留有练习，提出问题让你解决。不要跳过这些练习，很多知识都是在完成这些练习的过程中学到的。我们为每个练习都提供了答案，如果你实在做不出来（很容易被一些小问题卡住），看看答案也无妨！不过在看答案之前，还是要尽力先自己解决问题。而且在读下一部分之前，一定要确确实实地掌握前面的内容。

# 看这本书需要些什么：

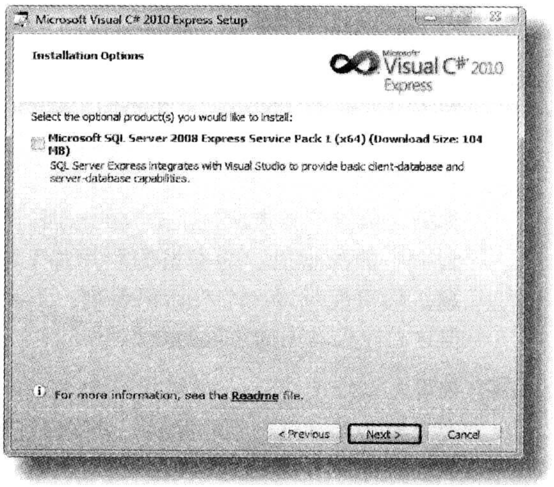
这本书使用的是Visual C# 2010 Express Edition。这说明，我们使用了C# 4.0和.NET Framework 4.0。本书中所看到的所有截屏图都取自这个版本，所以建议你也使用这一版本。如果你使用Visual Studio 2010 Professional、Premium、Ultimate或Test Professional版本，会看到一些细微的差别，这些我们都会尽可能特别指出。可以从微软的网站下载Express版本，即使机器上已经安装其他版本，甚至包括Visual Studio以前的版本，仍然还可以安装Express版本。

## 安装 VISUAL STUDIO 2010 EXPRESS EDITION

- 下载和安装Visual C# 2010 Express Edition非常容易。下面是Visual Studio 2010 Express Edition下载页面的链接：

  `http://www.microsoft.com/express/downloads/`

  要运行本书中的代码运行，并不需要将安装程序中的所有选项都选中，不过如果你愿意这么做也完全可以。



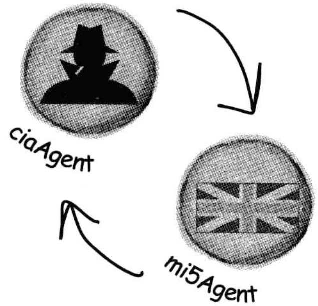

如果你确实必须使用一个老版本的Visual Studio、C#或.NET Framework，一定要记住，这本书中谈到的一些内容可能与你的版本不兼容。微软的C#小组为C#语言增加了一些相当酷的特性。千万要记住，如果你没有使用最新的版本，这本书里的一些代码有可能无法使用。

- 下载Visual C# 2010 Express Edition的安装包。要确保完全安装，应当安装所需的全部内容，包括：IDE（稍后将介绍）、.NET Framework 4.0以及其他工具。
- 一旦完成安装，会有一个新的“开始”菜单选项：Microsoft Visual C# 2010 Express Edition。单击这个菜单项将打开IDE，这就准备好了。

# 重要说明

要把这看做是一个学习过程，而不要简单地把它看成是一本参考书。我们在安排内容的时候有意做了一些删减，只要是对有关内容的学习有妨碍，我们都毫不留情地把这些部分一律删掉。另外，第一次看这本书的时候，要从第一页从头看起，因为书中后面的部分会假定你已经看过而且学会了前面的内容。

我们用了大量图表，使难懂的概念更易于理解。




**书里的实践活动不是可有可无的。**

这里的练习和实践活动不是可有可无的装饰和摆设。它们也是这本书核心内容的一部分。其中有些练习和活动有助于记忆，有些能够帮助你理解，还有一些对于如何应用所学的知识很有帮助。千万不要把这些练习跳过不做。只有池塘谜题不要求一定完成，不过通过这些池塘谜题，可以让你的大脑有机会考虑一些不太直接的小逻辑问题。

所有"Sharpen your pencil"活动都必须完成。




**我们有意安排了许多重复，这些重复非常重要。**

Head First系列的书有一个与众不同的地方，这就是，我们希望你确确实实地学会，另外希望在学完这本书之后你能记住学过了什么。大多数参考书都不太重视重复和回顾，但是由于这是一本有关学习的书，你会看到一些概念一而再、再而三地出现很多次。

**完成所有练习!**

写这本书时，我们做了一个重要的假设，假定你确实希望学习如何使用C#编程。所以我们认为你想马上动手，深入剖析代码。我们在每一章中都留有练习，提供了大量机会让你提高技艺。其中一些练习上会专门标有"动手做!"如果看到这个标志，说明我们会带领你一步步地解决一个特定问题。不过如果看到练习（Exercise）的logo标志（一双跑鞋），意味着这个问题的很大一部分都将留给你来解决，不过我们会给出提供的解决方案。不要害怕看答案，这不是偷看！不过，如果先努力自行解决问题，会学到更多东西。

标有Exercise（跑鞋）logo标志的实践活动非常非常重要！如果真的想好好学C#，就不要跳过这些练习。

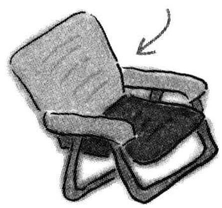



我们还把所有练习答案的源代码都放在了互联网上，供你下载。可以在`http://www.headfirstlabs.com/books/hfcsharp/`找到。

**"Brain Power（头脑风暴）"练习没有答案。**

有一些头脑风暴练习根本没有正确的答案，对于另外一些练习，头脑风暴实践活动中有一部分学习过程就是让你确定你的答案是否正确，以及在何种情况下正确。在其中一些头脑风暴练习中，你会得到一些提示，为你指明正确的方向。

如果看到Pool Puzzle（池塘谜题）标志，说明这个实践活动是可选的，如果你喜欢直截了当，不喜欢拐弯抹角，可能也不会喜欢这种练习。

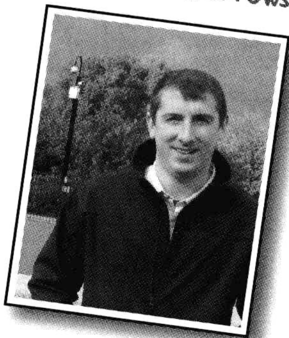


# 技术审校团队

Lisa Kellner



Chris Burrows



特别要感谢Chris敏锐的洞察力和极有帮助的反馈。

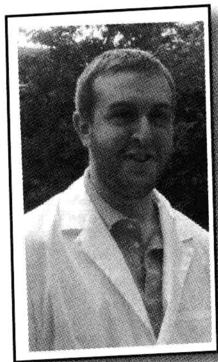
还有没有摆上照片（但同样让人钦佩的第一版审校人员）：Joe Albahari、Jay Hilyard、Aayam Singh、Theodore、Peter Ritchie、Bill Meitelski、Andy Parker、Wayne Bradney、Dave Murdoch和Bridgette Julie Landers。还要特别感谢Jon Skeet对第一版全面细致的审阅和建议！

Nick Paladino



David真是帮了大忙，特别是提供了一些非常巧妙的IDE技巧。

David Sterling



**致技术审校：**

写这本书时，里面有一大堆错误、问题、毛病、录入错误，还有糟糕的算术错误。当然，也许并没有那么差劲，不过还是非常感谢我们的技术审校为这本书付出的辛勤劳动。如果不是这个有史以来最强大的审校团队，这本书出版时肯定少不了错误（可能还包括一到两个相当严重的错误）……

首先，真心感谢Chris Burrows和David Sterling对我们详尽的技术指导。还要感谢Lisa Kellner，这是她为我们审校的第六本书，经她之手，最终作品的可读性大为改观。谢谢你，Lisa!还要特别感谢Nick Paladino，真是非常感谢！

Chris Burrows是微软C#编译器团队的开发人员，专门研究C# 4.0语言特性（尤其是动态特性）的设计和实现。

David Sterling参加Visual C#编译器团队已经近3年。

Nicholas Paldino自MVP计划开始之初就是微软.NET/C# MVP，在编程领域已经有超过13年的经验，特别是微软相关技术。

# 致谢

**致我们的编辑：**

感谢我们亲爱的编辑Brett McLaughlin和Courtney Nash，完成了这本书的编辑。Brett帮助我们完善了很多叙述，另外，第14章的漫画全是他的主意，我们认为这恰如其分地表达了我们的想法。多谢了！



Courtney Nash



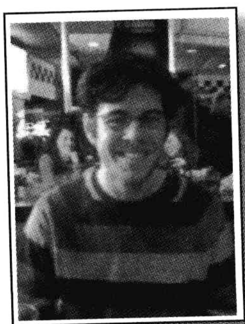
Brett McLaughlin

**致O'Reilly团队：**



Lou Barr

Lou Barr是一位无与伦比的图片设计师，她投入了大量时间，提供了一幅幅精美绝伦的图画。如果看到这本书里哪些部分棒极了，都要归功于她（还有她高超的InDesign技艺）。“实验室”部分所有怪物和外星人的图片都出自她之手，另外整个漫画书也全由她绘制。实在太感谢了，Lou!你是我们的英雄，向你致以发自内心的敬仰。



Sanders Kleinfeld

我们还要感谢O'Reilly的很多人，希望不要有任何遗漏。特别感谢策划编辑Rachel Monaghan，感谢编制索引的Lucie Haskins和Emily Quill，多亏她们严格的校对，感谢Ron Bilodeau投入的时间以及提供的专家意见，感谢Sanders Kleinfeld做了最后一遍全面检查。所有这些人的帮助，使这本书从最初策划终于按期交付出版。一如既往，太爱你了，Mary Treseler，真是等不及与你再次合作！还要衷心感谢另外一些朋友和编辑，Andy Oram和Mike Hendrickson。如果你现在正在读这本书，请感谢这个行业最杰出的宣传团队：Marsee Henon、Sara Peyton、Mary Rotman、Jessica Boyd、Kathryn Barrett，以及Sebastopol的其他人员。

# Safari®图书在线

Safari图书在线是一个按需提供资源的数字图书馆，从中可以很容易地搜索超过7500本技术书、参考书和视频，快速找到你想要的答案。

只需订购，就可以从我们的在线图书馆阅读任何页面，观看任何视频。你可以在你的手机和移动设备上看书，能够在新书出版之前获得书目，还可以享有特权了解正在编写的书稿并向作者提出反馈意见。你可以剪切粘贴代码示例、整理最喜欢的图书、下载你需要的章节、为重要内容设置书签、创建笔记、打印页面，此外还有大量节省时间的特性可以让你受益。

O'Reilly Media已经将这本书（英文版）上传到Safari图书在线服务。要想通过数字方式全面访问这本书以及O'Reilly和其他出版商提供的其他类似图书，可免费注册*http://my.safaribooksonline.com/?portal=oreilly*。

# 目录（概览）


| | | |
|---|---|---|
| | 引子 | xxix |
| 1 | C#助你快速开发：10分钟（甚至更短时间）内轻松搞定可视化应用 | 1 |
| 2 | 都只是代码：幕后的故事 | 41 |
| 3 | 对象：我们的方向！让代码更合理 | 85 |
| 4 | 类型与引用：现在是上午10：00，你知道你的数据在哪儿吗？ | 125 |
| | C#实验室1：赛狗日 | 169 |
| 5 | 封装：让你的隐私……属于你个人 | 179 |
| 6 | 继承：对象的家庭树 | 215 |
| 7 | 接口与抽象类：让类信守承诺 | 269 |
| 8 | 枚举与集合：存储大量数据 | 327 |
| | C#实验室2：冒险游戏 | 385 |
| 9 | 读/写文件：保存了字节数组，就留住了一切 | 407 |
| 10 | 异常处理：救火太老套 | 463 |
| 11 | 事件与委托：你不监视时代码在做什么 | 507 |
| 12 | 复习与预习：知识，能力，动手实践 | 541 |
| 13 | 控件与图片：应用的美化 | 589 |
| 14 | CAPTAIN AMAZING：对象之死 | 647 |
| 15 | LINQ：让数据在你的掌控之中 | 685 |
| | C#实验室3：入侵者 | 713 |
| i | 其他：本书最想介绍的11大内容 | 735 |


# 详细目录

## 引子

**让你的大脑来学C#** 你想坐下来学点东西，可是你的大脑却总在帮倒忙，一直在告诉你学这些不重要。你的大脑在想，“还是把空间留给更重要的事情吧，比方说要躲开哪些野兽，还有光着身子滑雪不太好吧。”那么你该如何骗过大脑，让它认为如果不学会C#你将无法生活？


| | |
|---|---|
| 谁适合看这本书？ | xxx |
| 我们也知道你的大脑在想什么 | xxxi |
| 元认知：有关思考的思考 | xxxiii |
| 可以用下面的方法让你的大脑就范 | xxxv |
| 看这本书需要些什么 | xxxvi |
| 重要说明 | xxxvii |
| 技术审校团队 | xxxviii |
| 致谢 | xxxix |

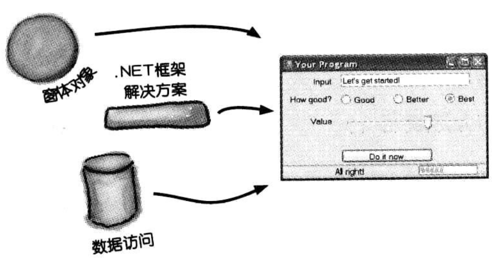
# 1 C#助你快速开发

## 10分钟（甚至更短时间）内轻松搞定可视化应用

**你想快速构建一流程序吗？**

有了C#，你就拥有了一种功能强劲的编程语言，这个意义非凡的工具将由你调遣。有了Visual Studio IDE，你不必再花好几个小时编写复杂的代码才能让按钮起作用。更棒的是，你可以集中精力完成你的具体工作，而不是去记住哪个方法参数对应按钮名，哪个参数对应按钮的标签。听起来是不是很吸引人？请翻开下一页，开始我们的编程之旅吧。





| | |
|---|---|
| 为什么你要学C# | 2 |
| C#和Visual Studio IDE使很多问题都变得容易 | 3 |
| 帮CEO实现无纸化 | 4 |
| 开始构建程序之前，先要了解用户的需求 | 5 |
| 你在Visual Studio中这样做…… | 8 |
| Visual Studio会为你这样做…… | 8 |
| 开发用户界面 | 12 |
| Visual Studio,在幕后 | 14 |
| 补充自动生成的代码 | 15 |
| 需要一个数据库存储信息 | 18 |
| IDE创建一个数据库 | 19 |
| SQL是数据库自己的语言 | 19 |
| 为Contact List应用创建数据库表 | 20 |
| 完成表的构建 | 25 |
| 将卡片数据插入到数据库 | 26 |
| 利用一个数据源连接窗体与数据库对象 | 28 |
| 为窗体增加数据库驱动控件 | 30 |
| 怎么把你的应用变成所有人的应用？ | 35 |
| 向用户提供应用 | 36 |
| 革命尚未成功：测试安装 | 37 |
| 你已经构建了一个完整的数据驱动应用 | 38 |

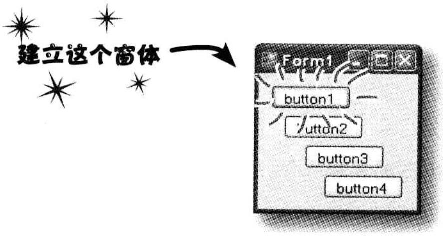
# 2 都只是代码

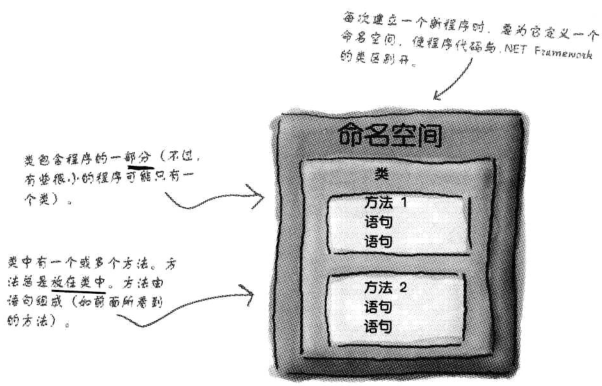
## 幕后的故事

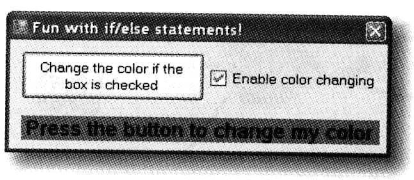
### 你是一个程序员，不只是一个IDE用户

用IDE可以做很多事情。不过不能过于夸大IDE，它的作用总是有限的。当然，构建一个应用时要完成大量重复的任务，而且IDE最擅长为你做这些事情。不过使用IDE还只是一个开始。可以让你的程序做更多的事情，唯一的办法就是编写C#代码。一旦掌握了编写代码的窍门，那么没有什么能难得住你的程序。


当你这么做时…… 42
……IDE会这么做 43
程序从哪里来 44
IDE帮你编写代码 46
在IDE中做修改，同时也正在修改你的代码 48
程序剖析 50
程序知道从哪里开始 52
两个类可以在同一个命名空间 59
程序使用变量处理数据 60
C#使用我们熟悉的数学符号 62
使用调试工具查看变量的变化 63
循环就是周而复始反复完成一个动作 65
开始编写代码 66
if/else语句做决策 67
建立条件，查看条件是否为true 68

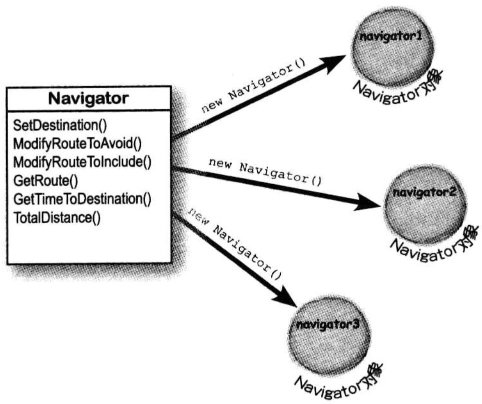



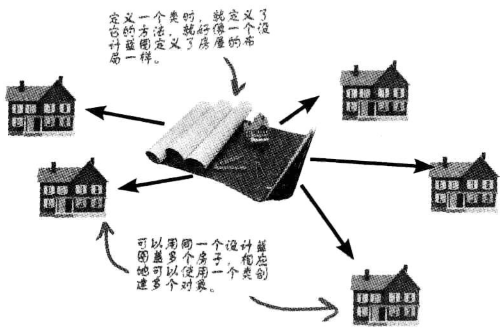
# 3 对象：我们的方向！

## 让代码更合理

**你写的每一个程序都要解决一个问题。**

构建一个程序时，首先考虑你的程序要解决什么问题，这通常是个不错的想法。这也说明了为什么对象非常有用。基于对象，你可以根据所要解决的问题建立代码的结构，把宝贵的时间用来考虑需要处理的具体问题，而不是深陷于编写代码的繁杂细节中。如果适当地使用对象，最后不仅写代码轻松，读代码也会很容易，另外还有利于修改代码。





| | |
|---|---|
| Mike怎么考虑他的问题 | 86 |
| Mike的汽车导航系统怎么考虑他的问题 | 87 |
| Mike的Navigator类有一些设置和改变路线的方法 | 88 |
| 使用前面所学构建一个使用类的程序 | 89 |
| Mike可以使用对象来解决他的问题 | 92 |
| 使用类来建立对象 | 93 |
| 由一个类创建新对象时，称为该类的一个实例 | 94 |
| 更好的解决方案……源于对象！ | 95 |
| 实例使用字段来跟踪状态 | 100 |
| 创建一些实例！ | 101 |
| 程序要做什么 | 103 |
| 使用合适的类名和方法名使代码更直观 | 104 |
| 为类提供一个自然的结构 | 106 |
| 类图可以帮助你组织类，让它们更有意义 | 108 |
| 构建一个类来处理人 | 112 |
| 为这些人创建一个工程 | 113 |
| 建立一个窗体与这些人交互 | 114 |
| 还有更容易的方法来初始化对象 | 117 |

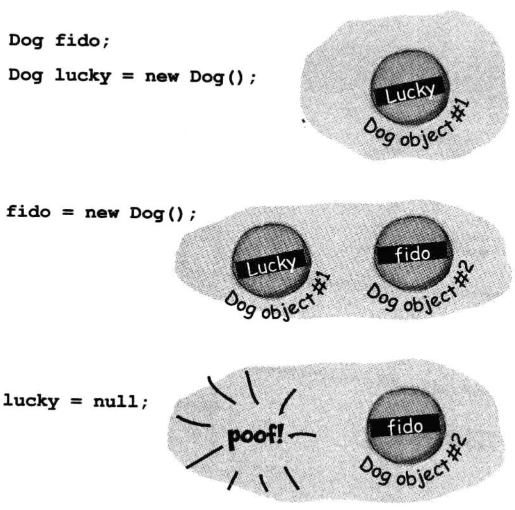
# 4 类型与引用

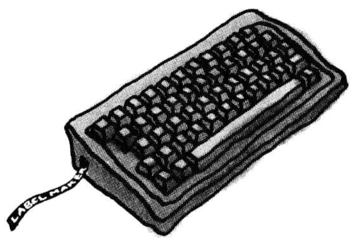
## 现在是上午10:00，你知道你的数据在哪儿吗？

**数据类型、数据库、军官数据……这些都很重要。**

如果没有数据，你的程序毫无用处。你需要用户提供的信息，要使用这些信息来查找或产生新的信息，再交给用户。实际上，编程中所做的几乎每一件事都是在以这样或那样的方式处理数据。在这一章中，你将全面了解C#的数据类型，学习如何在程序中处理数据，甚至还会发现关于对象的一些小秘密（知道吗？对象也是数据）。


| | |
|---|---|
| 变量的类型决定了它能存储哪种数据 | 126 |
| 变量就像数据外卖杯 | 128 |
| 10磅数据装进5磅的包里 | 129 |
| 即使数字的大小合适，也不能随意赋给任意变量 | 130 |
| 对一个过大的值完成强制类型转换时，C#会自动进行调整 | 131 |
| C#自动完成一些强制类型转换 | 132 |
| 调用一个方法时，实参必须与行参的类型兼容 | 133 |
| 结合=和操作符 | 138 |
| 对象也使用变量 | 139 |
| 用引用变量指示对象 | 140 |
| 引用就像对象的标签 | 141 |
| 如果不再有任何引用，对象就会被垃圾回收 | 142 |
| 多个引用及其副作用 | 144 |
| 两个引用意味着有两个途径来改变一个对象的数据 | 149 |
| 一种特殊情况:数组 | 150 |
| 欢迎品尝马虎Joe餐厅物美价廉的三明治 | 152 |
| 对象使用引用相互交谈 | 154 |
| 还没有对象 | 155 |
| 构建一个打字游戏 | 160 |

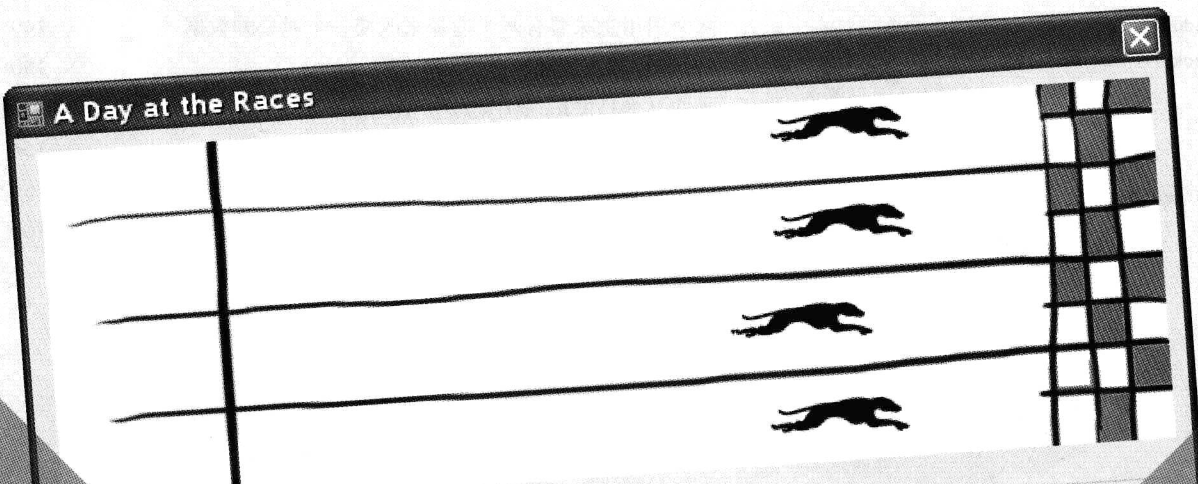



# C#实验室1

## 赛狗日

**Joe、Bob和Al热衷于赛狗，不过他们可不想把钱都输光。需要你为他们构建一个模拟系统，以便在下注之前知道谁是赢家。另外，如果你干得漂亮，他们还会给你分红。**

规范：构建一个赛狗模拟系统 170
最终产品 178



# 5 封装

## 让你的隐私……属于你个人

### 是不是想要更多点隐私？

有时你的对象也这么想。你肯定不希望不信任的人看你的杂志或者随意翻看你的银行票据。同你一样，好对象也不会让别的对象随意侵入它们的领地。这一章中，你将了解到封装的威力，你会把对象的数据设置为私有，并增加方法对这些数据的访问加以保护。



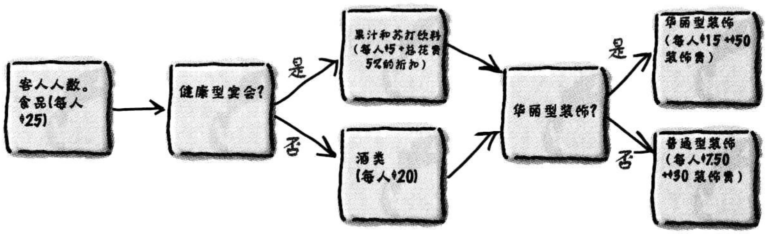








| | |
|---|---|
| Kathleen是一个策划人 | 180 |
| 预算工具要做些什么 | 181 |
| Kathleen的测试 | 186 |
| 每个选择都应当单独计算 | 188 |
| 很容易无意识地滥用对象 | 190 |
| 封装意味着保证类中的一些数据是私有的 | 191 |
| 使用封装来控制对类方法和字段的访问 | 192 |
| 但是realName字段真的得到了保护吗？ | 193 |
| 私有字段和方法只能从类的内部访问 | 194 |
| 封装保证数据干净 | 202 |
| 属性使封装更容易 | 203 |
| 构建一个应用测试Farmer类 | 204 |
| 使用自动属性完成这个类 | 205 |
| 如果想改变单位饲料数呢 | 206 |
| 使用构造函数初始化私有字段 | 207 |

# 6 继承

## 对象的家庭树

**有时你确实希望能像你的父母。**

你遇到过这样的对象吗？它几乎能完成你希望它做的所有工作，只是还差那么一点点。你是不是希望只是稍做改动这个对象就能完美无缺？我们说继承是C#语言中最为强大的概念和技术之一，以上只是其中的一个原因。读完这一章，你将了解如何派生一个对象来得到它的行为，同时还能保证灵活性，允许修改继承得到的行为。通过继承，就能避免重复代码，以更接近实际的方式对现实世界建模，最终开发的代码也将更易于维护。


Kathleen也承办生日聚会 216

需要一个BirthdayParty类 217

构建宴会应用2.0 218

类使用继承时，只需写一次代码 226

建立类模型：从一般到特定 227

怎样设计一个动物园模拟系统 228

使用继承避免子类中出现重复代码 229

想想如何对动物分组 231

创建类层次体系 232

每个子类都扩展其基类 233

子类可以覆盖方法来改变或替换它继承的方法 238

只要能使用基类，就可以使用它的某个子类 239

子类可以隐藏超类中的方法 246

使用override和virtual关键字来继承行为 248

现在准备完成Kathleen布置的任务 252

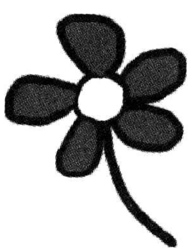
构建一个蜂巢管理系统 257

首先构建基本系统 258

使用继承扩展蜜蜂管理系统 263





# 7 接口与抽象类

## 让类信守承诺

**行动比言语更有力。**

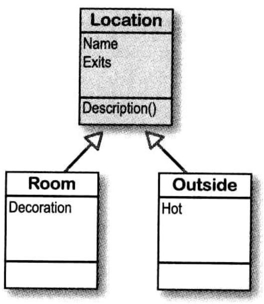
有时需要根据对象完成的工作将对象分组，而不是根据它们继承的类来分组。这就引入了接口，通过接口，可以使用任何一个能完成任务的类。但是能力越大，责任也越大，实现了一个接口的类必须承诺履行它的所有职责……否则编译器就会抱怨，明白吗？

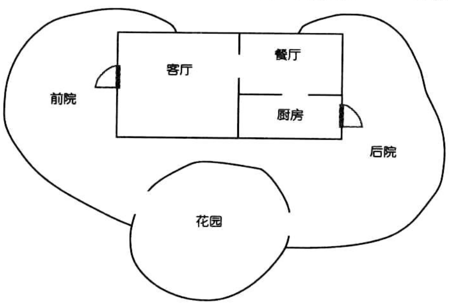
继承

抽象

封装

多态





再来看蜂巢问题 270

可以使用继承为不同类型的蜜蜂创建相应的类 271

接口告诉类必须实现某些方法和属性 272

使用interface关键字定义接口 273

实现接口的类必须包括接口的所有方法 275

不能实例化接口，不过可以引用接口 278

接口引用类似于对象引用 279

可以使用“is”查看一个类是否实现了某个接口 280

接口可以继承其他接口 281

对象和接口的向上强制转换 285

向下强制转换允许你把家用电器变回咖啡机 286

接口也可以向上和向下强制转换 287

不只是公共和私有 291

访问修饰符改变可见性 292

有些类不应实例化 295

抽象类就像类和接口之间的一个过渡 296

抽象方法没有方法体 299

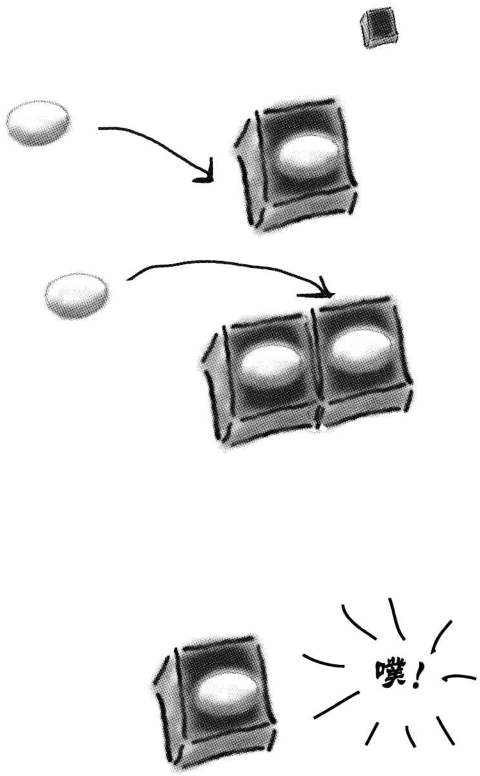
多态表示一个对象可以有多种形态 307

# 8 枚举与集合

## 存储大量数据

**简直是倾盆大雨。**

在真实世界里，我们处理的数据绝对不会是一点点。实际上，数据会成包、成捆，甚至成堆地压过来。我们需要一些非常强大的工具来整理所有这些数据，这就引入了集合。利用集合，可以存储程序需要处理的所有数据，并进行排序和管理。这样一来，你可以专心考虑编写程序处理数据，而由集合负责为你记录数据。





字符串并不总适用于存储所有类型的数据 328
Enum枚举允许处理一组合法值 329
通过枚举，可以用名来表示数字 330
可以使用数组来创建一副牌…… 333
List比数组更灵活 336
泛型可以存储任何类型 340
集合初始化方法类似于对象初始化方法 344
下面来创建一个Duck列表 345
列表很容易，但排序可能很困难 346
IComparable<Duck>帮助列表完成鸭子排序 347
使用IComparer告诉List如何排序 348
创建比较对象的一个实例 349
IComparer可以完成复杂的比较 350
覆盖ToString()方法让对象描述自己 353
更新foreach循环，让Duck和Card能自行输出 354
可以使用IEnumerable向上强制转换整个列表 356
可以建立你自己的重载方法 357
字典功能通览 364
构建一个使用字典的程序 365
还有更多集合类型…… 377
队列是FIFO —— 先进先出 378
栈是LIFO —— 后进先出 379

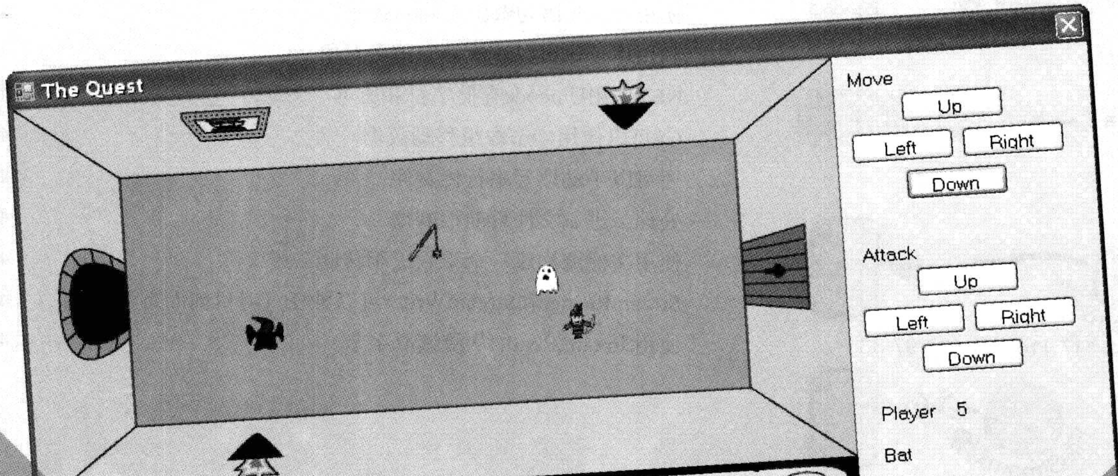
# C#实验室2

## 冒险游戏

**你的任务是构建一个冒险游戏，一个勇猛的冒险家在探险，一关一关地打败那些危险的敌人。你要构建一个按回合进行的系统，也就是说，玩家走一步，然后敌人走一步。玩家可以移动或攻击，接下来各个敌人得到机会移动并攻击。游戏会一直进行下去，直到玩家打败7关的所有敌人，或者被敌人打死。**


规范: 构建一个冒险游戏 386

好戏就要开始了 406

# 9 读/写文件

## 保存了字节数组，就留住了一切

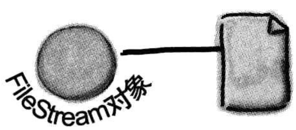
**有时多点持久性是有回报的。**

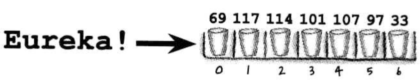
到目前为止，所有程序的生命都很短暂。它们启动后，运行一段时间，然后就会关闭。但这往往不够（特别是处理非常重要的信息时）。需要能够把你的工作保存下来。在这一章中，我们将介绍如何向文件写数据，然后如何从文件读回这个信息。你将了解.NET流类，还会掌握十六进制和二进制的一些奥秘。


.NET使用流读写数据 408
不同的流读写不同的数据 409
FileStream向文件读写字节 410
只用3个简单步骤就能向文件写文本 411
使用两个对象读写 415
数据可以经过多个流 416
使用内置对象弹出标准对话框 419
对话框也是.NET控件 420
对话框也是对象 421
IDisposable确保对象适当地撤销 427
利用using语句避免文件系统错误 428
写文件通常要做大量决策 434
使用switch 语句做出正确的选择 435
利用串行化可以一次读写整个对象 442
.NET使用Unicode存储字符和文本 447
C#可以使用字节数组移动数据 448
还可以手动读写串行化文件 451
处理二进制文件可能很麻烦 453
使用文件流构建一个十六进制转储工具 454
StreamReader和StreamWriter可以胜任（对目前而言） 455
使用Stream.Read()从流读取字节 456



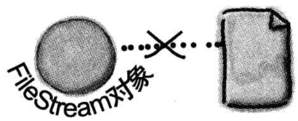










# 10 异常处理

## 救火太老套

**程序员可不是消防员。**

你夜以继日地努力工作，遍查各种技术手册，并通读几本绝妙的Head First书，终于攀上了你职业的高峰，成为一名首席程序员，不过还是经常因为你的程序崩溃或者不像预期的那么好而被频频叫回。如果希望编程不那么单调乏味，最好的做法就是去修正一个奇怪的bug……不过，利用异常处理，可以编写代码来处理可能出现的问题。更妙的是，甚至可以一方面处理这些问题，另一方面让程序继续运行。


Brian希望他的借口能移动 464

程序抛出一个异常时，.NET会生成一个 Exception对象 468

所有异常对象都继承自Exception 472

调试工具可以帮助跟踪和避免代码中的异常 473

使用IDE的调试工具准确找出借口管理系统中哪里出了问题 474

用try和catch处理异常 479

想调用的方法有风险会导致什么结果? 480

使用调试工具完成try/catch流程 482

如果有些代码总要运行，可以使用一个finally块 484

一个类抛出异常，另一个类捕获异常 491

蜜蜂需要一个OutOfHoney异常 492

避免大量问题的一种简便方法:
利用using，可以轻松实现try和finally 495

避免异常:实现IDisposable完成自己的清理 496

史上最糟糕的catch块：全能型+注释 498

临时方案是允许的(权宜之计) 499

关于异常处理的一些简单想法 500

Brian终于能度假了…… 505



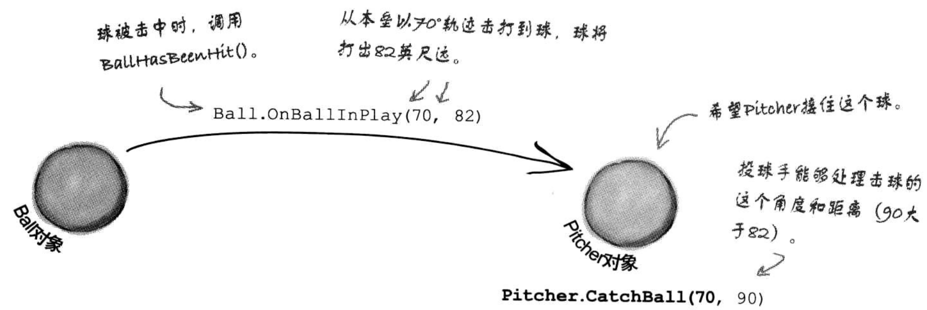

# 11 事件与委托

## 你不监视时代码在做什么

**对象开始自己考虑问题。**

你并不总能控制对象在干什么。有时候……会发生一些事情。如果真是如此，你希望对象能足够聪明，对发生的事情做出响应。这正是事件要做的事情。一个对象发布一个事件，其他对象订购这个事件，大家联合起来保证工作顺利进行。听起来很不错，不过如果希望对象对于谁能监听有所控制，就不那么容易了。这里就需要用到回调。


| | |
|---|---|
| 希望你的对象自己考虑问题吗? | 508 |
| 不过一个对象怎么知道要做出响应呢? | 508 |
| 出现一个事件时……对象会监听 | 509 |
| 然后，其他对象处理这个事件 | 511 |
| 把各部分连接起来 | 512 |
| IDE会为你自动创建事件处理程序 | 516 |
| 通用EventHandler允许你定义自己的事件类型 | 522 |
| 之前创建的窗体都使用了事件 | 523 |
| 一个事件，多个处理程序 | 524 |
| 连接事件发送者和事件接收者 | 526 |
| 委托代表一个具体的方法 | 527 |
| 委托的实际演练 | 528 |
| 对象可以订购事件…… | 531 |
| 使用回调控制谁在监听 | 532 |
| 回调只是一种使用委托的方式 | 534 |

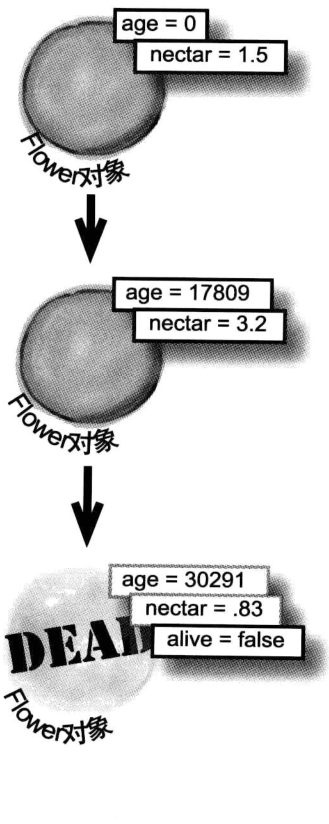
# 12 复习与预习

## 知识，能力，动手实践

**光学不练是没有意义的，还需要真正动手去实践。**

除非你自己动手写一些真正能运行的代码，否则很难相信你能真正掌握C#中那些深奥的概念。这一章中，我们将使用前面学到的知识动手实践，还会预习后面很快要讲到的一些新知识。我们会构建一个相当复杂的应用，确保你真正掌握了前面各章学到的内容。所以，做好准备……现在就来构建真正的软件。

花的生与死





| | |
|---|---|
| 你已经学了不少了，伙计 | 542 |
| 我们还是不错的养蜂人 | 543 |
| 蜂巢模拟系统体系结构 | 544 |
| 构建蜂巢模拟系统 | 545 |
| 花的生与死 | 549 |
| 现在需要一个Bee类 | 550 |
| P.A.H.B.(程序员与无家可归的蜜蜂) | 554 |
| 蜂巢靠蜂蜜运转 | 554 |
| 填写Hive类 | 558 |
| 蜂巢的Go()方法 | 559 |
| 准备构建World | 560 |
| 我们要构建一个基于回合的系统 | 561 |
| World的代码 | 562 |
| 提供蜜蜂的行为 | 568 |
| 主窗体告诉世界调用Go()推进 | 570 |
| 可以使用World得到统计信息 | 571 |
| 定时器反复触发事件 | 572 |
| 下面处理蜜蜂分组 | 580 |
| 用集合来收集……数据 | 581 |
| 利用LINQ，可以很容易地处理数据库和集合中的数据 | 583 |
| 最后一个难题:打开和保存 | 585 |

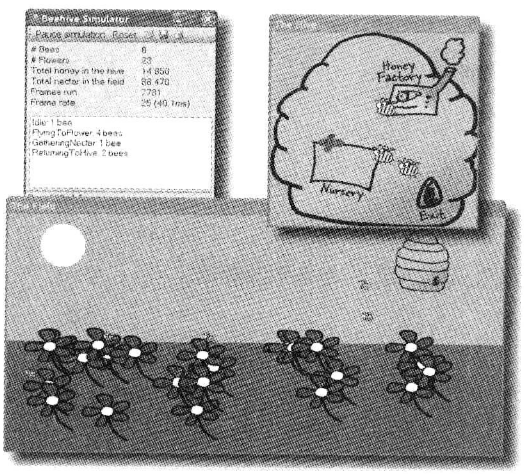
# 13 控件与图片

## 应用的美化

**有时需要你自己控制图形化。**

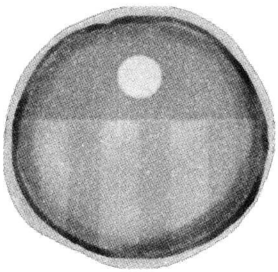
大多数情况下，我们都依赖于控件处理应用的可视化部分。不过有时这还不够，比如说，如果想对一个图片完成动画，就需要自己来控制。一旦开始实现动画，就要为.NET程序创建你自己的控件，可能需要增加双缓冲，甚至直接在窗体上绘图。一切都要从Graphics对象和Bitmap开始，还要有决心摒弃既有的图形化处理。

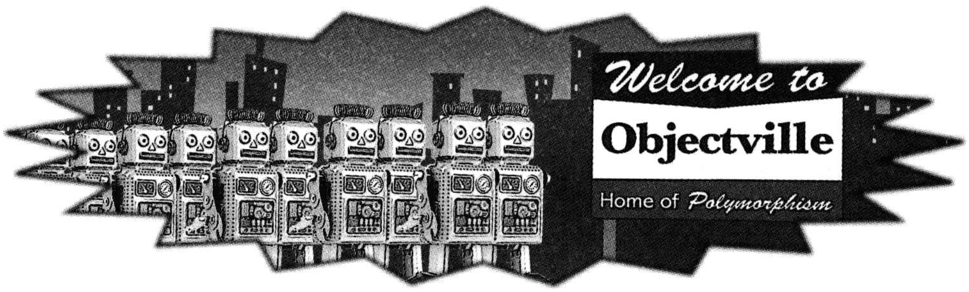

| | |
|---|---|
| 你一直在使用控件与程序交互 | 590 |
| 窗体控件也只是对象 | 591 |
| 使用控件完成蜂巢模拟系统的动画 | 592 |
| 在体系结构中增加Renderer | 594 |
| 控件很适合表示可视化显示元素 | 596 |
| 构建第一个动画控件 | 599 |
| 创建一个按钮向窗体增加BeeControl | 602 |
| 控件还需要撤销它们的控件！ | 603 |
| UserControl是构建控件的简便方法 | 604 |
| 模拟系统的Renderer使用BeeControl在窗体上绘制动画蜜蜂 | 606 |
| 为工程增加蜂巢和花场窗体 | 608 |
| 构建Renderer | 609 |
| 使用一个Graphics对象调整Bitmap大小 | 618 |
| 图像资源存储在Bitmap对象中 | 619 |
| 使用System.Drawing自行控制图形化 | 620 |
| GDI+图形化30秒之旅 | 621 |
| 使用Graphics在窗体上绘制图片 | 622 |
| Graphics可以修正透明问题…… | 627 |
| 使用Paint事件固定图形 | 628 |
| 进一步分析窗体和控件如何自行重绘 | 631 |
| 双缓冲使动画看起来更平滑 | 634 |
| 使用Graphics对象和事件处理程序完成打印 | 640 |

# 14 CAPTAIN AMAZING 对象之死


你的最后机会……对象的最终化方法 654
最终化方法到底何时运行? 655
Dispose()处理using, 最终化方法处理垃圾回收 656
最终化方法不能依赖于稳定性 658
让对象在Dispose()中自行串行化 659
struct看起来像是一个对象…… 663
……但不是对象 663
复制值，指定引用 664
栈与堆:再谈内存 667
用out参数使方法返回多个值 670
使用ref修饰符按引用传递 671
使用可选参数设置默认值 672
需要不存在的值时可以使用可为空的类型 673
可为空的类型可以让程序更健壮 674
Captain Amazing……还不够 677
扩展方法为现有类增加新行为 678
扩展一个基本类型: string 680

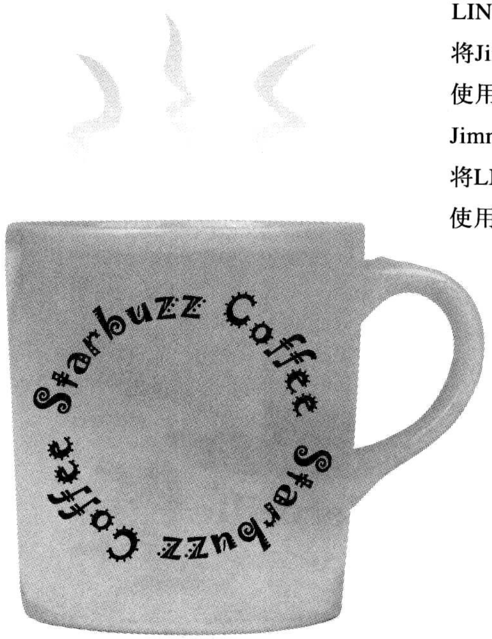


# 15 LINQ

## 让数据在你的掌控之中

**这是一个数据驱动的世界……最好知道如何在这个世界上生存。** 也许原先你可以编写几天甚至几个星期的程序而不需要处理大量数据，这样的日子已经一去不复返了。如今，一切都离不开数据。实际上，往往不只需要处理一处的数据……另外通常还会采用多种格式。数据库、XML、其他程序中的集合……这些都是一个优秀C#程序员需要面对的任务。这里就可以用到LINQ。LINQ不仅允许你采用一种简单直观的方式查询数据，还可以对数据分组，以及合并不同数据来源的数据。


一个简单的项目…… 686
……但是到处都是数据 687
LINQ可以从多个来源取出数据 688
已经为LINQ建立了.NET集合 689
利用LINQ可以轻松地查询 690
LINQ很简单，但是你的查询不一定简单 691
LINQ是个多面手 694
LINQ可以将结果合并分组 699
将Jimmy漫画书的价格合并分组 700
使用join将两个集合合并到一个查询 703
Jimmy省了一大笔钱 704
将LINQ连接到一个SQL数据库 706
使用一个join查询连接Starbuzz和Objectville 710

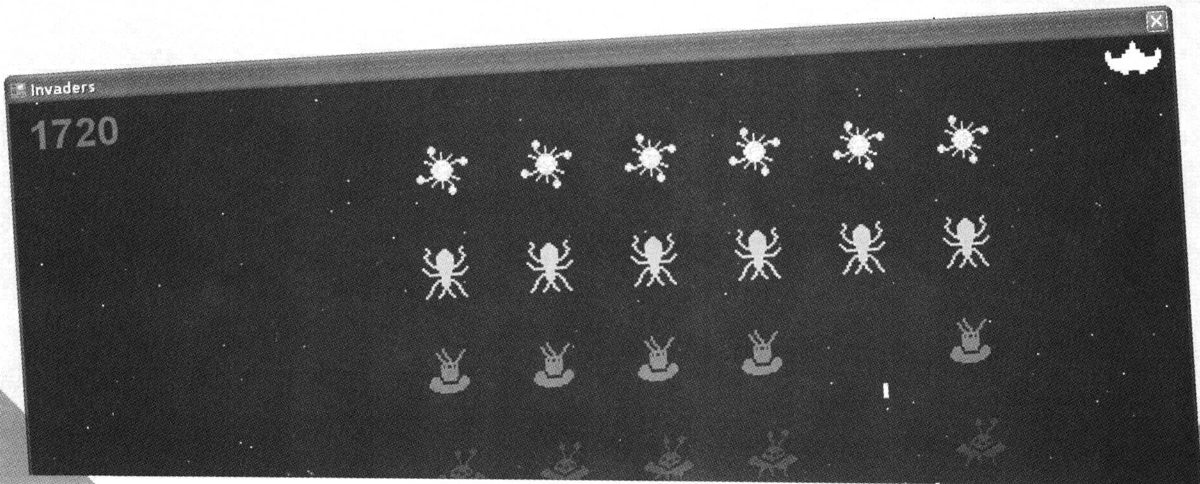



# C# 实验室 3

## 入侵者

**在这个实验室里，你将回顾视频游戏历史上曾经最游行、最受人推崇的游戏之一，这个游戏不需要多做介绍了。下面来构建这个鼎鼎大名的入侵者游戏——Invaders。**

经典的视频游戏 714
还有很多可以做…… 733



# 其他

## i 这本书最想介绍的11大内容

### 好戏才刚刚开始！

我们已经介绍了很多相当棒的工具，可以帮助你使用C#构建非常强大的软件。不过，这本书绝对无法涵盖每一个工具、技术或技巧，实在是篇幅不够。我们必须做出艰难的抉择，究竟包含哪些内容，而哪些内容无法涵盖。下面这一些内容就无法涵盖在内。不过尽管书中没有介绍，但是我们还是认为这些内容非常重要，也很有用，所以希望你们能有一些基本的认识。


#1. 基础知识 736

#2. 命名空间和程序集 742

#3. 使用BackgroundWorker让你的UI能够响应 746

#4. Type类和GetType() 749

#5. 相等性、IEquatable和Equals() 750

#6. 使用yield return创建可枚举的对象 753

#7. 重构 756

#8. 匿名类型、匿名方法和lambda表达式 758

#9. 使用DataContractSerializer串行化数据 760

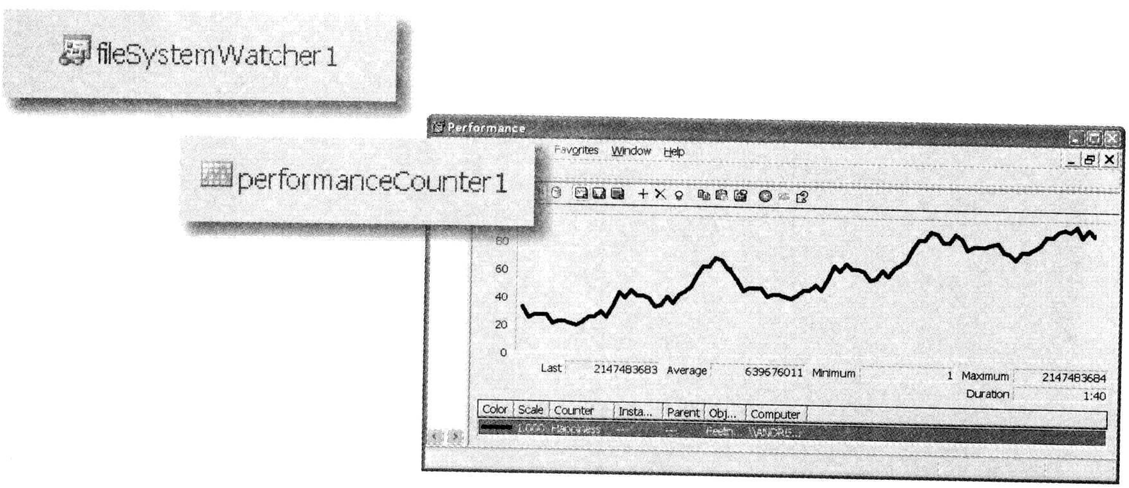
#10. LINQ to XML 762

#11. Windows表现基础库 764

你知道C#和.NET Framework能够…… 766







# 1 C#助你快速开发

# 10分钟（甚至更短时间）内轻松搞定可视化应用




## 你想快速构建一流程序吗？

有了C#，你就拥有了一种功能强劲的编程语言，这个意义非凡的工具将由你调遣。有了Visual Studio IDE，你不必再花好几个小时编写复杂的代码才能让按钮起作用。更棒的是，你可以集中精力完成你的具体工作，而不是去记住哪个方法参数对应按钮名，哪个参数对应按钮的标签。听起来是不是很吸引人？请翻开下一页，开始我们的编程之旅吧。

# 为什么你要学C#

利用C#和Visual Studio IDE，你能轻松而快速地编写代码。使用C#时，IDE是你最好的朋友和最忠实的伙伴。

IDE或Visual Studio集成开发环境（Integrated Development Environment）是使用C#编程的一个重要部分。这个集成开发环境可以帮助你编辑代码、管理文件以及发布工程。

## IDE会自动为你完成……

每次着手编写一个程序时，或者只是在窗体上放一个按钮时，你的程序中都需要大堆的重复代码。

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace A_New_Program
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

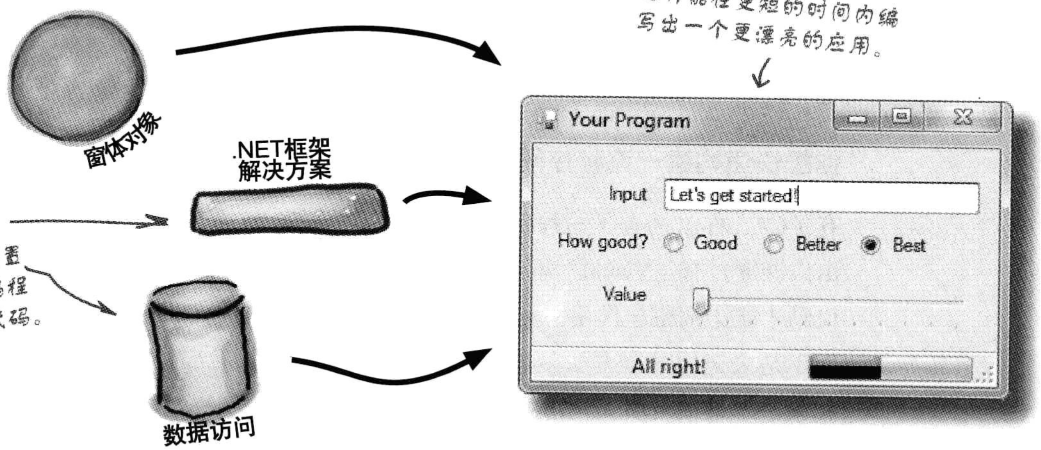
```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(105, 56);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(292, 267);
    this.Controls.Add(this.button1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
```

只是在窗体上画一个按钮就需要这么多代码。要向窗体增加更多可视化元素，需要的代码可能是这里的10倍。

## 利用Visual Studio和C#能得到……

利用类似C#的Windows编程语言，再结合Visual Studio IDE，你就能立即集中精力实现程序的具体功能：

这样能在更短的时间内编写出一个更漂亮的应用。




C#、.NET框架和Visual Studio IDE提供了一些预置结构，可以处理大多数编程任务中都会出现的烦琐代码。

# C#和Visual Studio IDE使很多问题都变得容易

使用C#和Visual Studio时，不必做任何额外的工作就能得到所有这些优秀特性。总结起来，利用它们可以做到：

1. 快速地构建一个应用。使用C#编写程序简直是小菜一碟。这种语言功能相当强大，而且学起来非常容易。另外，Visual Studio IDE还会自动为你做一些工作。可以把日常的编程任务交给IDE完成，而你专心考虑如何实现所需的具体功能。

2. 设计美观的用户界面。Visual Studio IDE中的Form Designer是目前使用最方便的设计工具之一。它的功能极其丰富，你会发现，建立让人赞叹的用户界面正是C#应用开发中最令人满意的一个方面。利用Visual Studio IDE，可以构建功能完备的专业程序，而不必花数小时完全从头开始编写一个图形化用户界面。

3. 创建数据库并与数据库交互。IDE为构建数据库提供了一个易于使用的接口，并且可以与SQL Server Compact Edition，以及多种其他流行的数据库系统无缝集成。

4. 专注于解决你的实际问题。IDE会帮你做很多工作，但是使用C#实现什么功能仍然要由你来控制。IDE只是让你能更专注于你的程序、你的工作（或娱乐），以及你的客户。不过，IDE确实会处理所有烦琐的工作，例如：

    - ★ 跟踪维护所有工程
    - ★ 使你能轻松地编辑工程的代码
    - ★ 跟踪维护工程中的图像、音频、图标和其他资源
    - ★ 管理数据库并与之交互

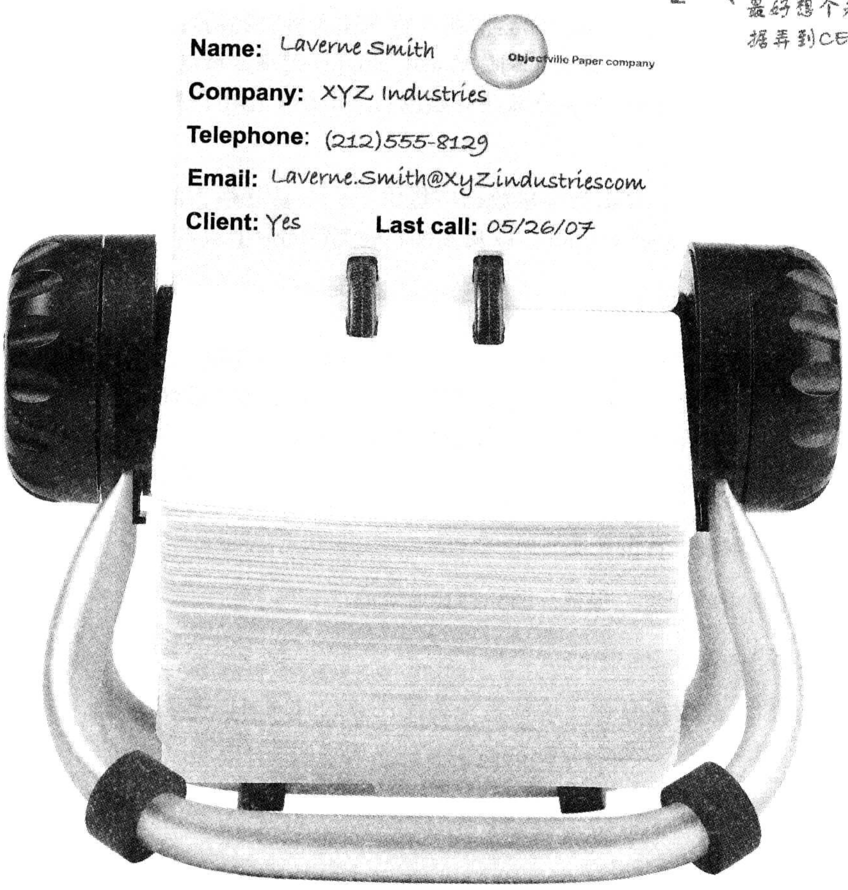
    所有这些意味着，原本需要用来完成这些常规编程任务的时间可以节省下来，而用来构建真正一流的程序。

接下来你会了解这是什么意思。

# 帮CEO实现无纸化

Objectville纸业公司刚刚聘请了一位新的CEO。他喜欢徒步旅行，喝咖啡，非常热爱大自然……所以决定为保护森林贡献一点力量。他希望自己成为一个无纸化主管，首先从他的联系卡片入手。这周末他要前往阿斯彭滑雪，希望等他回来能看到一个新的地址簿程序。否则…… 很难讲…… 他可不像原来那个CEO那么心慈手软（原先的CEO现在沦落到正在找工作）。

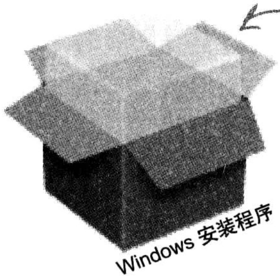
最好想个办法尽快把这些数据弄到CEO的笔记本电脑里。



# 开始构建程序**之前**，先要了解用户的需求

着手编写这个地址簿应用（或者任何其他应用）之前，需要先要用点时间考虑一下：谁将使用这个应用，另外他们需要从这个应用得到什么。

① CEO希望在办公室计算机和笔记本电脑上都能运行这个地址簿程序。他需要一个安装程序，确保两台机器上都能安装有关的全部文件。




CEO希望在他的桌面计算机和笔记本电脑上都能运行这个程序，所以必须要有一个安装程序。

**开始编写代码之前先要考虑用户和他们的需要，这样等你完成时，他们才会对最终产品满意！**

② Objectville纸业公司销售部门也希望访问CEO的地址簿。他们可以利用这些数据建立邮件列表，得到用户反馈从而进一步促进纸产品销售。

CEO认为，要让公司里的每一个人都能看到他的数据，最好的办法就是建立一个数据库，这样一来，只需维护所有联系数据的一个副本。

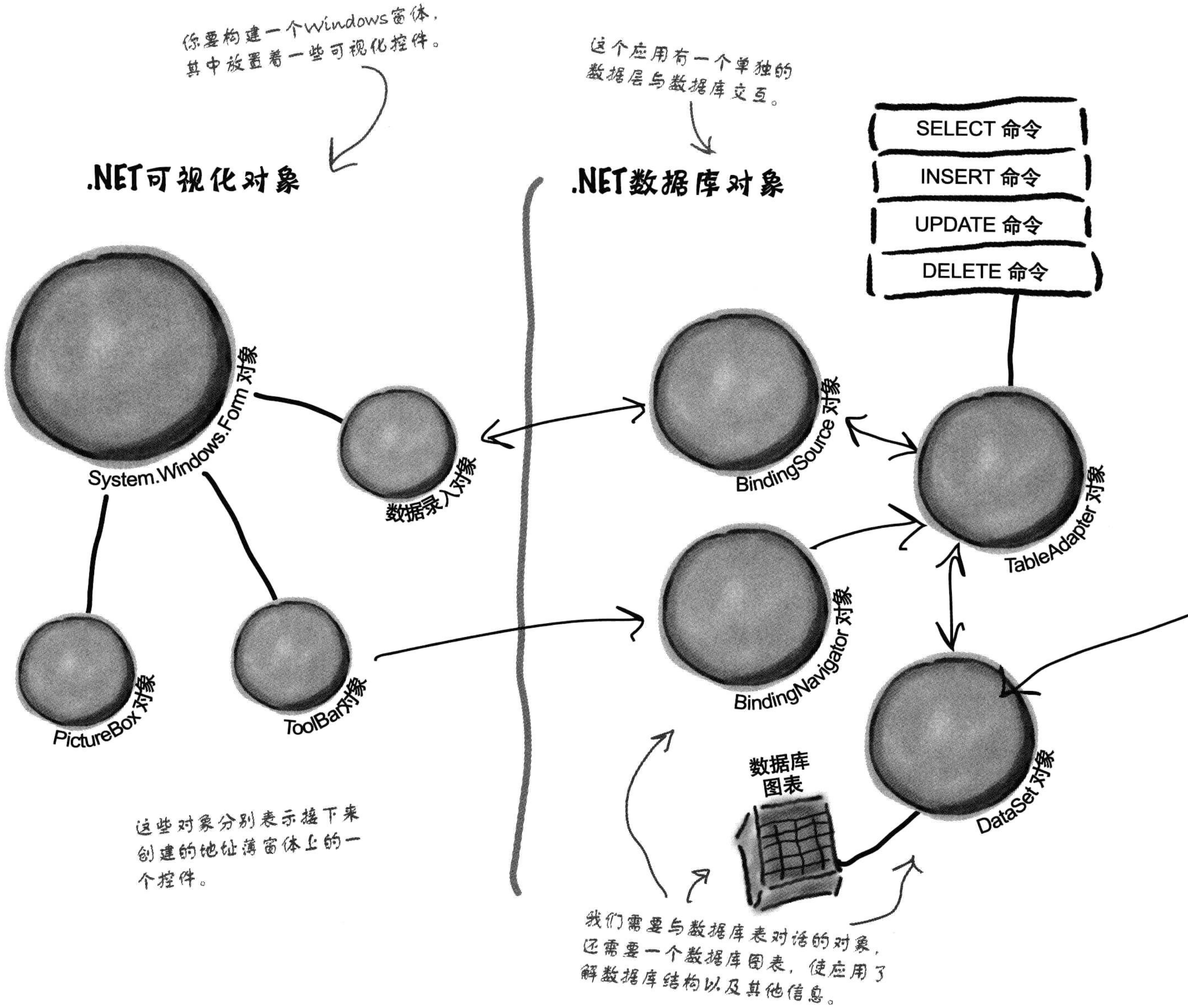


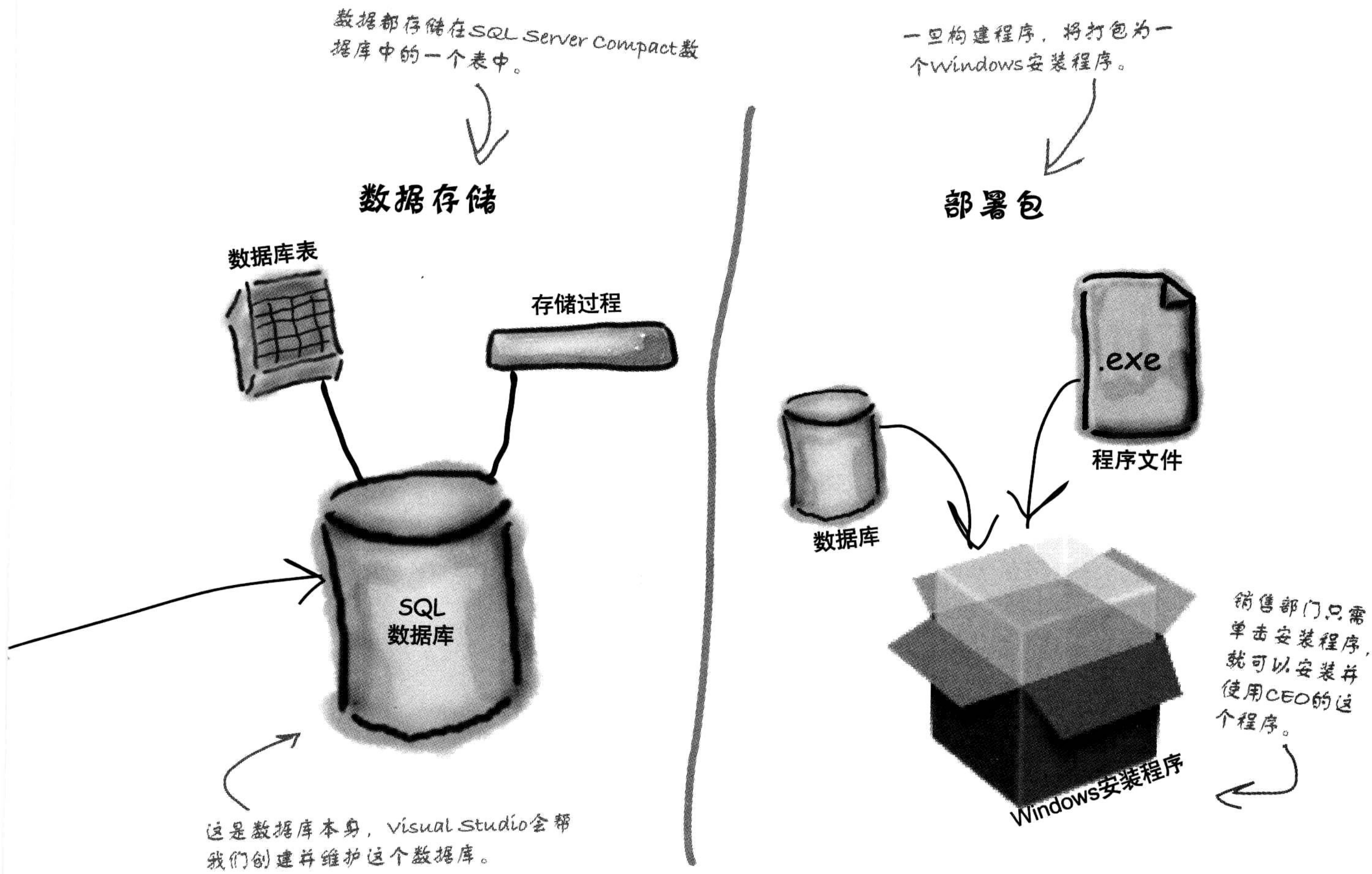

我们已经知道，利用Visual C#，处理数据库将相当容易。通过在数据库中维护联系信息，尽管只有一个数据副本，但CEO和销售部门都能访问这个信息。

# 以下是你要实现的目标

需要一个有图形用户界面的应用、与数据库交互的对象、数据库本身，还要有一个安装程序。听上去工作很多，不过经过后面几页的介绍，你能轻松地完成所有这些工作。

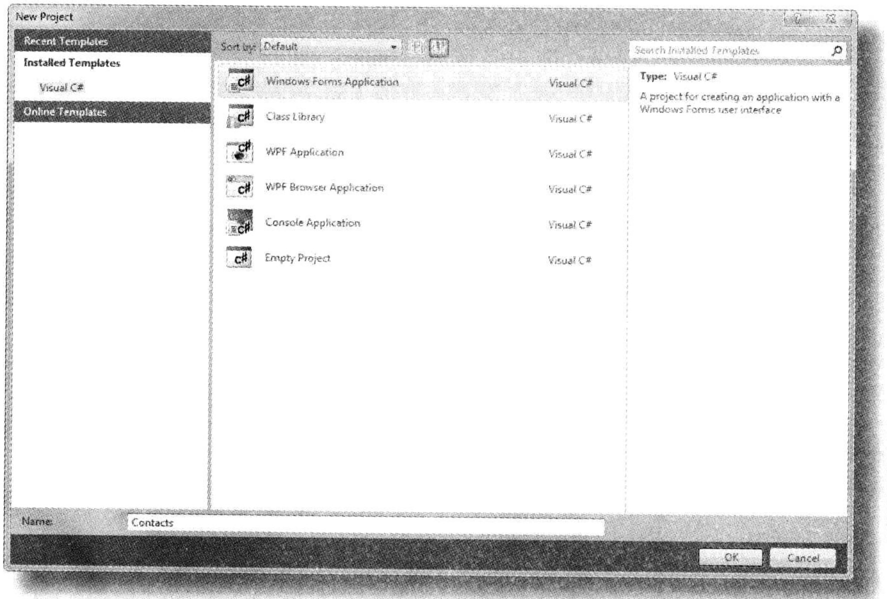
以下是我们要创建的程序的结构：




数据都存储在SQL Server Compact数据库中的一个表中。
数据存储
数据库表
存储过程
SQL
数据库
这是数据库本身，Visual Studio会帮我们创建并维护这个数据库。
一旦构建程序，将打包为一个Windows安装程序。
部署包
.exe
程序文件
数据库
Windows安装程序
销售部门只需单击安装程序，就可以安装并使用CEO的这个程序。

## 你在Visual Studio中这样做……

如果还没有启动Visual Studio，把它打开。跳过开始页，从File菜单选择New Project。将工程命名为“Contacts”，再单击OK。有多种工程类型可以选择。这里选择Windows Forms Application，然后选择新工程名“Contacts”。







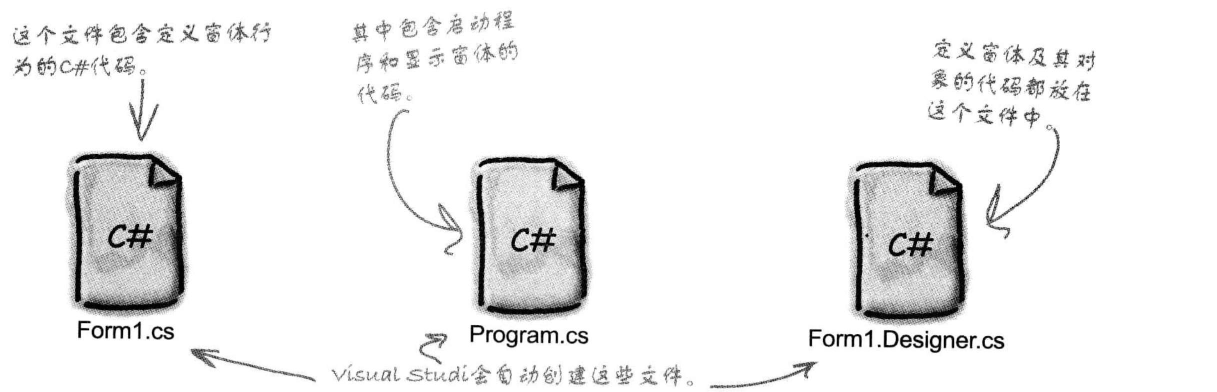
**你的IDE中的显示可能会稍有不同。**

这是Visual Studio 2010 Express Edition中显示的“New Project”窗口。如果你使用的是Professional或Team Foundation版本，可能有一点儿差别。不过不用担心，所有工作都是一样的。

## Visual Studio会为你这样做……

创建一个新工程时，一旦保存这个工程，IDE会分别创建`Form1.cs`、`Form1.Designer.cs`和`Program.cs`文件。它将这些文件增加到Solution Explorer窗口，另外默认地，这些文件将放到`My Documents\Visual Studio 2010\Projects\Contacts\`目录中。

一旦创建工程，一定要记着保存，可以从File菜单选择“Save All”，这样会把该工程的所有文件保存到文件夹。如果选择“Save”，只会保存目前正在处理的那个文件。



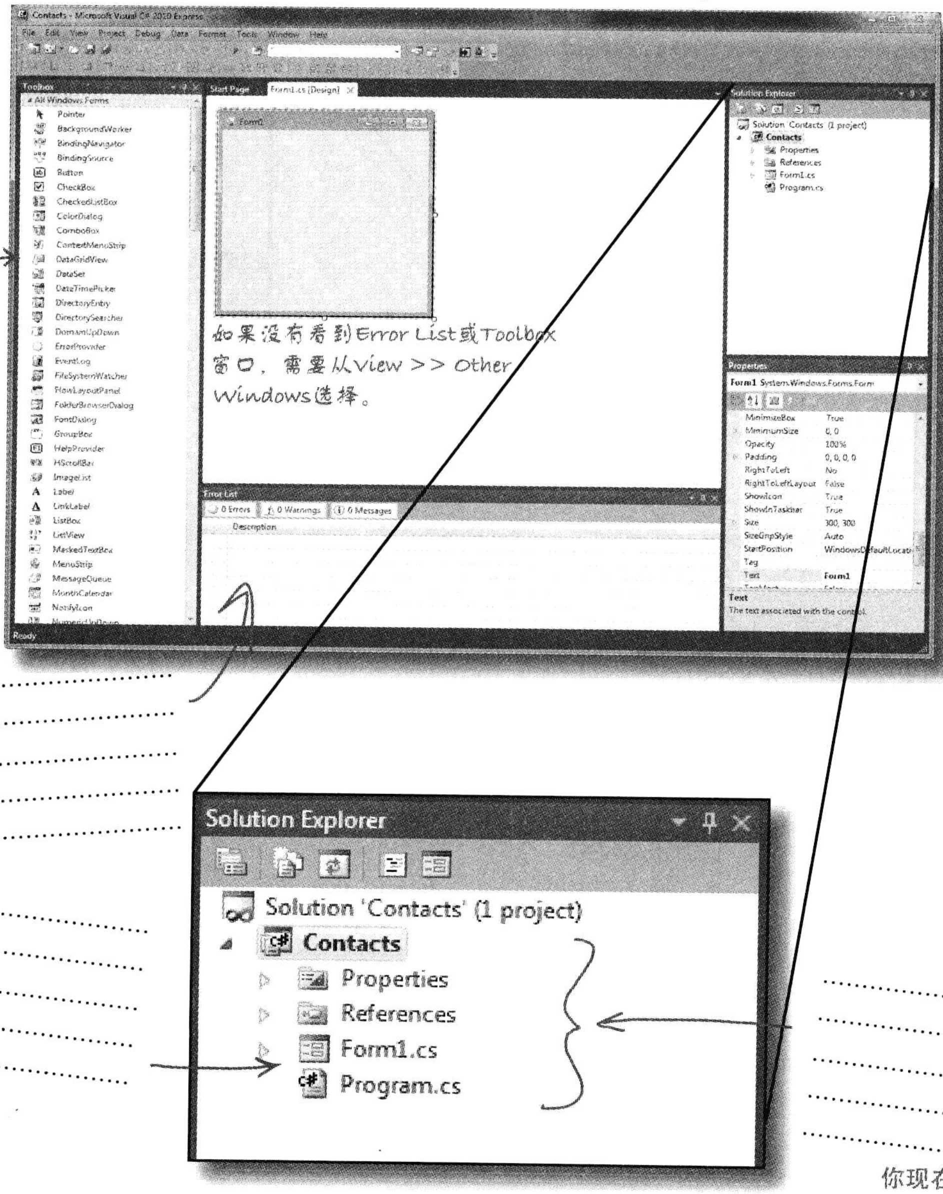
## Sharpen your pencil

现在你的屏幕可能如下。根据前面已经了解的知识，你应该能得出以下大部分窗口和文件是什么。一定要从View → Other Windows菜单选择Toolbox和Error List，将这两个窗口打开。然后尝试在每个空栏上填入一个注解，指出IDE的这一部分是做什么的。我们已经填写了其中的一个空栏，其余的由你完成。

这个工具条上有一些按钮，可以应用到IDE中目前正在处理的内容。

如果你的IDE看上去与这个图不大一样，则可以从Window菜单选择“Reset Window Layout”。

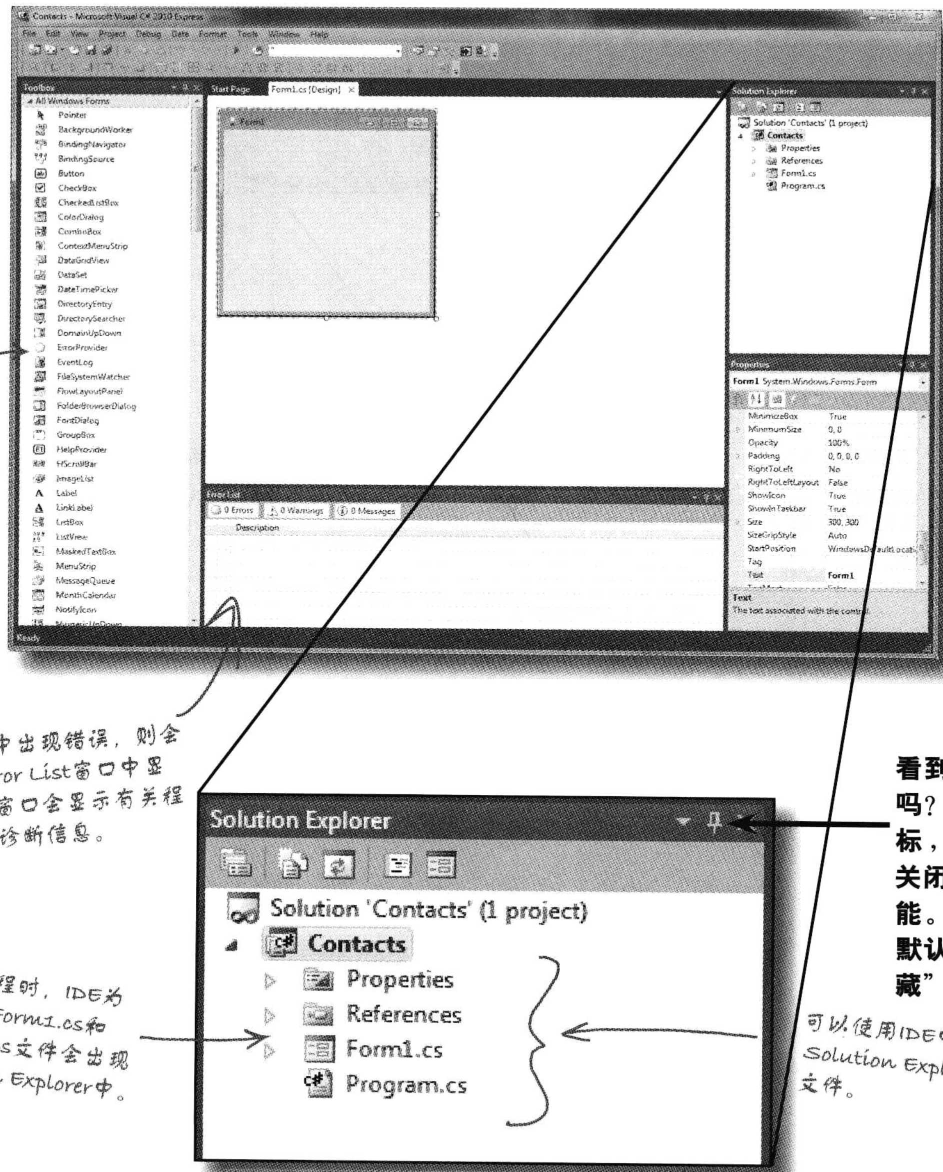
我们在下面放大了这个窗口，来留出更大空间让你填写答案。




如果没有看到Error List或Toolbox窗口，需要从View >> Other Windows选择。

# Sharpen your pencil Solution

我们已经填入Visual Studio C# IDE各个部分的注解。也许你写的答案与这里不同，不过起码最基本的应该明确：各个窗口是什么，另外IDE的各个部分做什么用。



## there are no Dumb Questions

**问：** **既然IDE能为我编写所有这些代码，学C#是不是就是学习怎么使用IDE？**

**答：** 错。IDE的优点在于，它能自动地为你生成一些代码，但是它的作用仅此而已。IDE在某些方面确实很擅长，比如能为你建立很好的起点，能够自动地改变窗体上控件的属性等。不过，对于编程中最困难的部分，也就是确定你的程序要做什么，以及怎么让它做到，IDE就无能为力了。尽管Visual Studio IDE是目前最高级的开发环境之一，但它的能力仍然有限。要由你（而不是IDE）来编写具体完成工作的代码。

**问：** **我在Visual Studio里创建了一个新工程，不过在My Documents下的“Projects”文件夹里怎么找不到它呢？怎么回事？**

**答：** 在Visual Studio 2010中，第一次创建一个新工程时，IDE会把这个工程创建到你的Local Settings\Application Data\Temporary Projects文件夹里。第一次保存工程时，会提示你指定一个新文件名，然后把它保存到My Documents\Visual Studio 2008\Projects文件夹。如果你试图打开一个新工程，或者关闭临时工程，就会提示你保存或者放弃这个临时工程（注意：Visual Studio的其他非Express版本并不使用临时工程文件夹。那些版本会直接在Projects文件夹中创建工程）。

**问：** **如果IDE创建的代码并不是我的工程里所需要的，该怎么办呢？**

**答：** 这些代码都可以改变。IDE会根据最常用的拖动或增加元素的方式来创建代码。不过，有时这可能并不是你所期望的。IDE为你做的所有一切，包括它创建的每一行代码，增加的每一个文件，都是可以修改的，可以直接手动地编辑文件，也可以通过IDE中简便好用的界面来完成修改。

**问：** **只是下载并安装Visual Studio Express就可以吗？要实现这本书介绍的全部内容，有必要使用某个不免费的Visual Studio版本吗？**

**答：** 这本书里的所有内容都可以利用免费的Visual Studio版本（可以从Microsoft网站下载）完成。Express和其他版本（Professional和Team Foundation）之间的差别并不影响编写C#代码和创建功能完备的应用。

**问：** **我可以改变IDE生成的文件的名字吗？**

**答：** 当然可以，创建一个新工程时，IDE会提供一个默认的窗体，名为Form1（相应的文件名为Form1.cs、Form1.Designer.cs和Form1.resx）。不过，只要你愿意，完全可以使用Solution Explorer改变这些文件名。默认地，文件名与窗体名相同。如果改变文件名，则会看到Properties窗口中窗体名仍为Form1。可以在Properties窗口中修改“(Name)”行来改变窗体名。如果这样做，文件名就不会随之改变。C#并不关心你为文件或窗体（或者程序中的任何其他部分）选择什么名字，不过这里确实有一些需要遵守的规则。然而如果选择了合适的名字，这会让你的程序更易于处理。对于现在来说，先不用担心名字，我们还会在后面更详细地讨论如何为程序中的各个部分选择好名字。

**问：** **我现在正在察看IDE，不过我的屏幕看上去与你的不太一样！有些窗口没有，另外一些窗口的位置也不对。怎么回事？**

**答：** 如果单击“Window”菜单下的“Reset Window Layout”，则IDE会为你恢复默认的窗口布局。然后你可以用“View → Other Windows”菜单让你的屏幕与这一章给出的屏幕相同。

**Visual Studio会为你生成一些代码，可以将这些代码作为应用的起点。**

**要保证应用完成所需的功能，这仍然完全由<u>你</u>负责。**

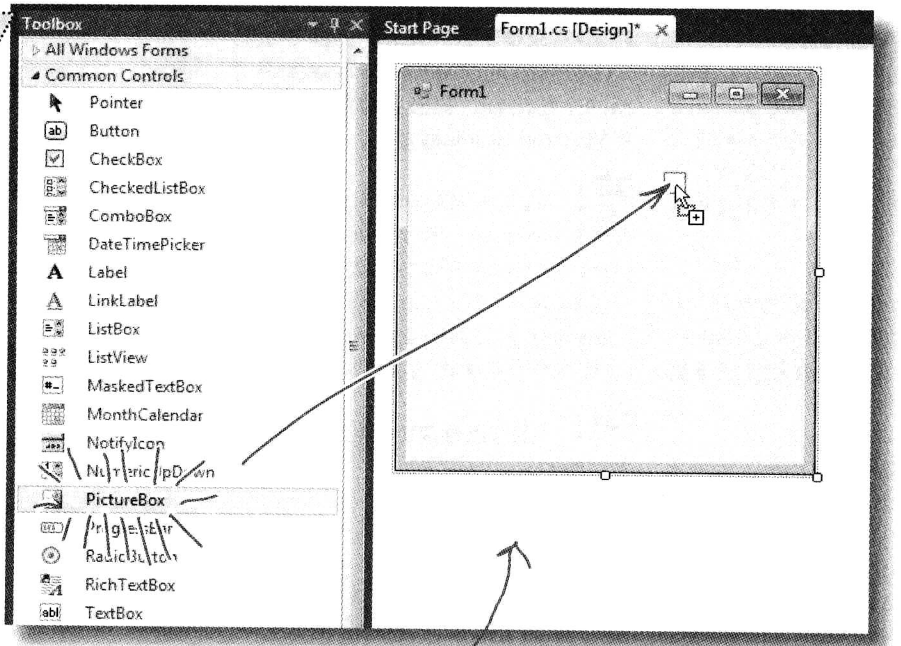
# 开发用户界面

增加控件和美化用户界面非常容易，只需在Visual Studio IDE中拖放控件就可以办到。下面向窗体增加一个logo标志：

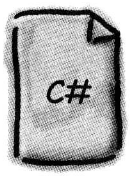
❶ **使用PictureBox控件增加一个图片。**

单击工具条中的PictureBox控件，把它拖到你的窗体中。在后台，IDE会在Form1.Designer.cs中增加这个新图片控件的相应代码。

如果没有看到工具箱，则可以把鼠标停在IDE左上角显示的文字“Toolbox”上，这就会展开工具箱。如果没有显示“Toolbox”，从View菜单选择“Toolbox”，就能看到了。




每次在窗体上对控件属性做出改变时，IDE就会对Form1.Designer.cs中的代码做相应调整。

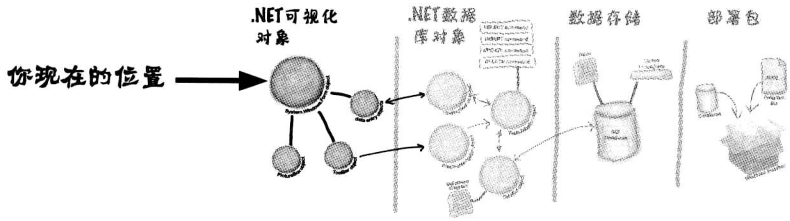



Form1.Designer.cs



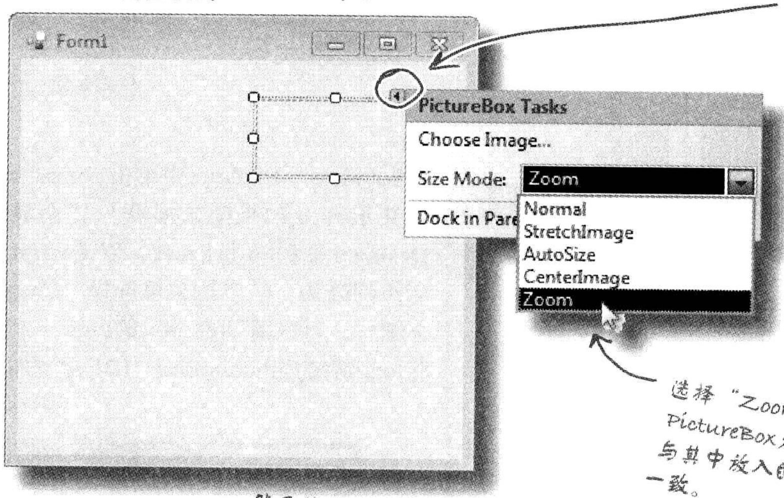

**即使你不擅长用户界面设计也无大碍。**

后面还会相当详细地介绍怎样设计优秀的用户界面。现在只要把logo标志和其他控件放在窗体上就行了，先不必考虑具体行为。后面还将增加一些样式。



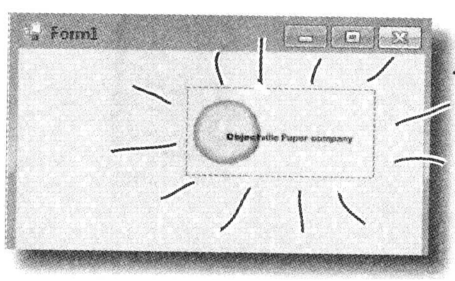

❷ **设置PictureBox为Zoom模式。**

窗体上的所有控件都有一些属性可以设置。单击对应一个控件的小黑箭头就能访问这些属性。为了说明这是如何做到的，下面将PictureBox的Size属性改为“Zoom”：：




❸ **下载Objectville纸业公司logo。**

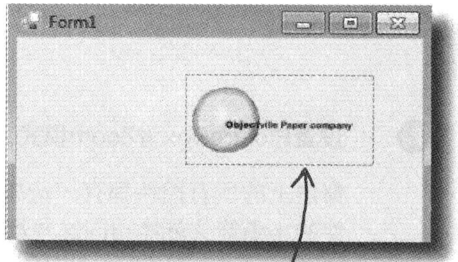
从Head First实验室（**http://www.headfirstlabs.com/books/hfcsharp**）下载Objectville纸业公司的logo，并保存到你的硬盘上。然后单击PictureBox属性箭头，选择Choose Image。你会看到弹出一个Select Resources窗口，单击“Local Resource” 单选钮来启用窗体上方的“Import…” 按钮。单击这个按钮，找到下载的logo，就大功告成了。



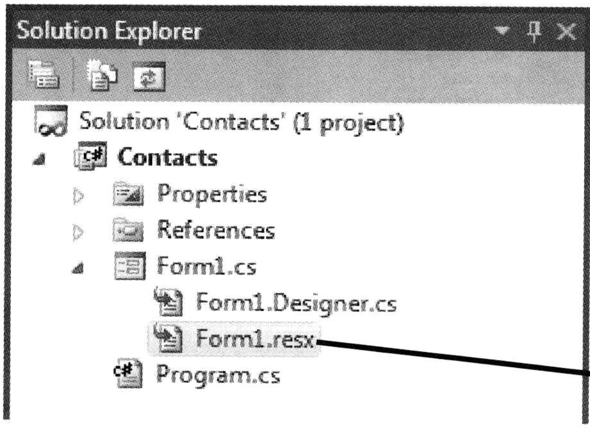
# Visual Studio，在幕后

每次在Visual Studio IDE中做一件事时，IDE都会为你编写代码。创建logo并告诉Visual Studio使用你下载的这个图像时，Visual Studio会创建一个资源，并把它关联到你的应用。资源（resource）可以是任何图形文件、音频文件、图标或者可以打包到应用中的其他类型数据文件。这个图形文件会集成到程序中，所以在另一个计算机上安装这个程序时，这个图形文件也会随之安装，以便PictureBox使用。

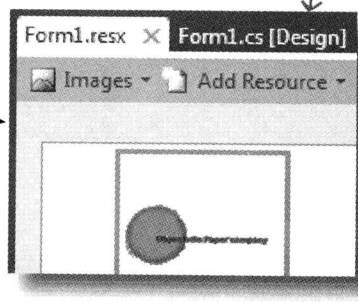
把PictureBox控件拖到窗体上时，IDE会自动创建一个名为`Form1.resx`的资源文件来存储这个资源，并在工程中维护。双击这个文件，可以看到这个新导入的图像。



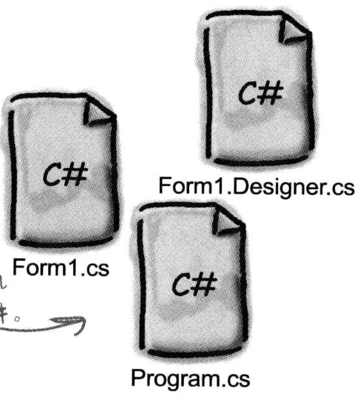

这个图像现在是Contact List应用中的一个资源。



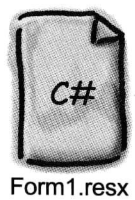

在Solution Explorer中单击`Form1.cs`旁边的“展开”图标将其展开（如果尚未展开），会显示两个文件：Form1.Designer.cs和Form1.resx。双击Form1.resx，单击“Strings”旁边的箭头，从下拉菜单选择“Images”（或者按Ctrl+2组合键），可以看到你导入的logo。正是这个Form1.resx文件将logo链接到PictureBox，IDE会增加代码来建立这个链接。




如果从上一页Select Resource对话框选择另一个“Import”按钮，你的图像会出现在Solution Explorer的Resources文件夹中。不用担心，只需回到Select Resources对话框，选择“Local Resource”，再把这个图像重新导入到资源中，它就会出现在这里了。



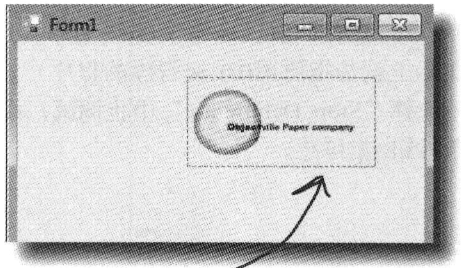

这些是之前Visual Studio创建的文件。




导入图像时，IDE会为你创建这个文件。其中包含与Form1关联的所有资源（图形、视频、音频和其他数据）。

# 补充自动生成的代码

IDE已经为你创建了大量代码，不过你还可以深入这些代码，做一些补充。下面为logo增加一个功能：用户运行程序并双击logo时将显示一条About消息。




在IDE中编辑窗体时，双击任何工具箱控件都会导致IDE在工程中自动增加代码。确保窗体在IDE中显示，然后双击PictureBox控件。IDE会向工程中增加代码，每次用户单击PictureBox时都会运行这个代码。应该可以看到弹出以下代码：

```
public partial class Form1 : Form
{

   public Form1()
   {

       InitializeComponent();

   }

   private void pictureBox1_Click(object sender, EventArgs e)

   {

       MessageBox.Show("Contact List 1.0.\nWritten by: Your Name", "About");

   }
}
```

双击PictureBox控件时，IDE会创建这个方法。运行应用时，每次用户单击logo都会运行这个方法。

从方法名可以清楚地看出这个方法何时运行：有人单击这个PictureBox控件时就会运行此方法。

双击PictureBox时，会打开这个代码，而且有一个光标在这个位置上闪烁。键入具体的代码过程中，忽略IDE弹出的所有窗口。尽管这是IDE有心提供帮助，不过现在我们还不需要。

输入这行代码，会导致弹出一个消息框，其中包含你提供的文本。这个消息框标题为"About"。

一旦输入这行代码，应使用IDE工具条上的Save图标或者从File菜单选择"Save"保存。要养成选择"Save All"保存的好习惯！

## there are no Dumb Questions

**问：** 什么是方法？

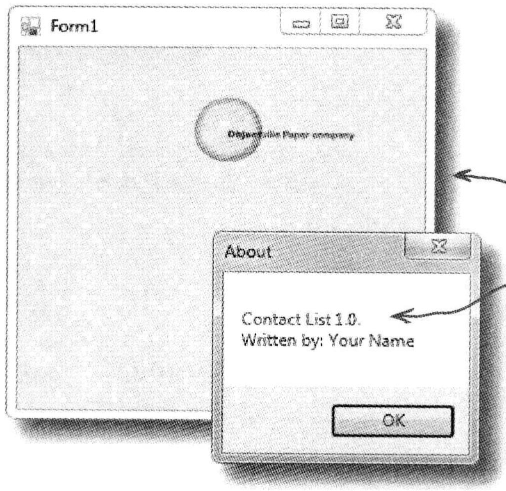
**答：** 方法就是一个有名字的代码块。我们会在第2章更多地讨论方法。

**问：** \n是做什么的？

**答：** 这是一个换行符。它告诉C#把"Contact List 1.0."放在一行上，再重新开始一行放上"Written by:"。

## 已经可以运行你的应用了

按键盘上的F5键，或者单击工具条上的绿色箭头按钮（▶），看看目前已经做了哪些工作（这称为“调试”，实际上就是指使用IDE运行你的程序）。可以从Debug菜单选择“Stop Debugging”中止调试，或者单击这个工具条按钮■停止。



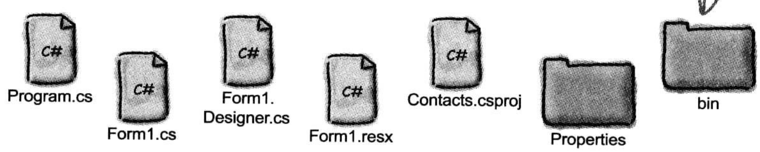

这3个按钮都能正常工作，而你并不需要为此编写任何代码。

单击OPC logo会弹出你刚编码实现的About消息框。

## 我的文件到哪去了？

运行程序时，Visual Studio会把你的所有文件都复制到`My Documents\Visual Studio 2010\Projects\Contacts\Contacts\bin\debug`。甚至可以直接切换到那个目录，双击IDE创建的.exe文件来运行你的程序。

C#将你的程序转换为一个可以运行的文件，称为可执行文件。可以在这里的debug文件夹中找到这个可执行文件。




这没有错。文件夹有两个层次。内部文件夹包含具体的C#代码文件。

### there are no Dumb Questions

**问：在我的IDE中，绿箭头标为“Debug”（调试），这有问题吗？**

**答：** 没问题。调试就是指在IDE中运行你的应用（至少对于目前来说是这样）。后面我们还会更多地讨论调试，不过目前可以简单地把它认为是运行程序的一种方法。

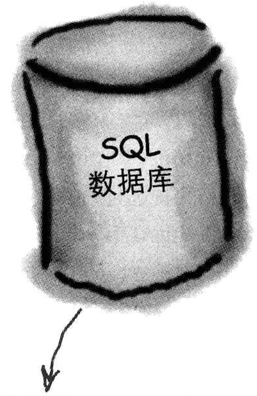
**问：在我的工具条上没有看到Stop Debugging按钮。怎么回事？**

**答：** Stop Debugging按钮只会出现在一个特殊的工具条上，只有当程序正在运行时才会显示这个工具条。重新启动应用，看看它会不会出现。

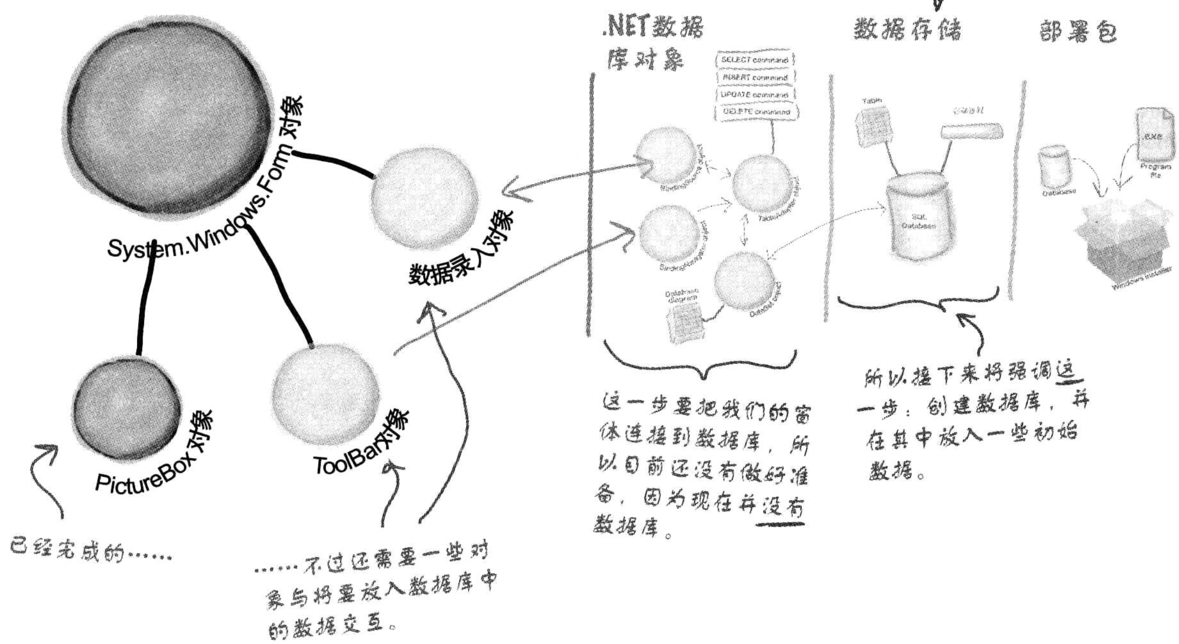
## 目前为止所做的工作

我们已经构建了一个窗体，并创建了一个PictureBox对象，单击这个控件时会弹出一个消息框。接下来，需要增加联系卡片上的所有其他数据域，比如联系人的名字和电话号码。

下面把这些信息存储在一个数据库中。Visual Studio能帮我们把各个数据域直接连接到这个数据库，这意味着我们不需要那么麻烦地编写大量数据库访问代码（这一点真的很棒）。不过，要做到这一点，需要先创建我们的数据库，这样窗体上的控件才能与之关联。因此，我们将从.NET可视化对象部分跳到数据存储部分。




### .NET可视化对象




**Visual Studio能够生成代码将窗体连接到一个数据库，不过生成这个代码之前首先需要有一个数据库才行。**

# 需要一个数据库存储信息

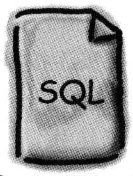
向窗体增加其余数据域之前，需要先创建一个将与窗体关联的数据库。IDE会创建大量代码将我们的窗体与数据连接，不过首先我们需要定义数据库本身。

做下面的工作之前必须先中止调试。

**1 为工程增加一个新的SQL数据库。**

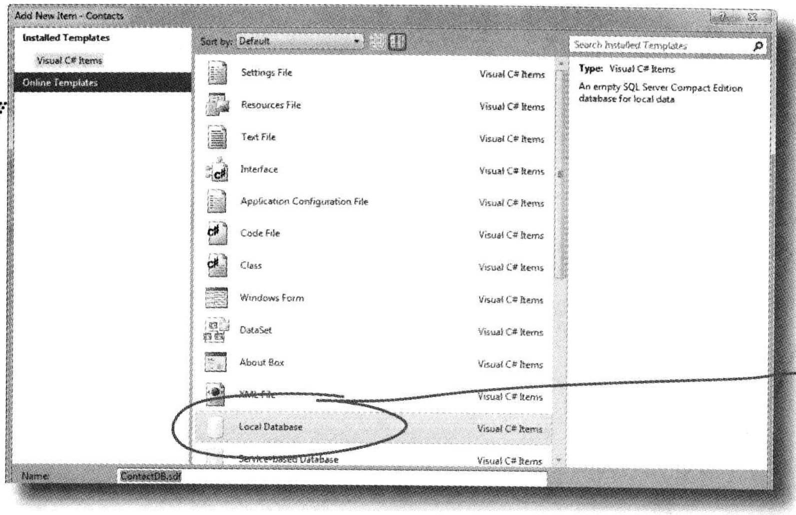
在Solution Explorer中，右键点击Contacts工程，选择Add，然后选择New Item。再选择SQL数据库图标，命名为ContactDB.sdf。

这个文件就是我们的新数据库。




ContactDB.sdf

选择Local Database创建一个SQL Server Compact Edition文件，其中包含你的整个数据库。将这个文件命名为ContactDB.sdf。




本地数据库（Local Database）实际上就是一个SQL Server Compact Edition数据库文件，通常扩展名为SDF。它提供了一个便捷的途径，从而可以在程序中嵌入一个数据库。

**2 单击Add New Item中的Add按钮。**

**3 数据源配置向导。**

现在我们不想配置数据源，所以单击Cancel按钮。等建立了数据库结构之后再来完成配置。

**4 在Solution Explorer中查看数据库。**

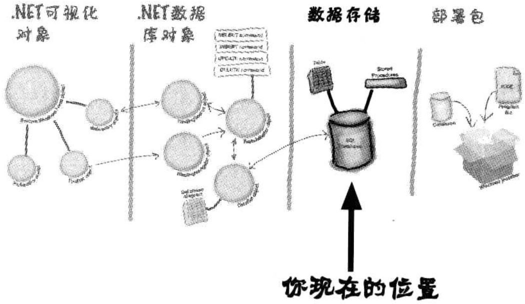
查看Solution Explorer，会看到ContactDB已经增加到文件列表中。在Solution Explorer中双击ContactDB.sdf，查看屏幕的左边。现在工具箱已经变成一个Database Explorer。




**如果使用的不是Express版本，看到的将是“Server Explorer”而不是“Database Explorer”。**

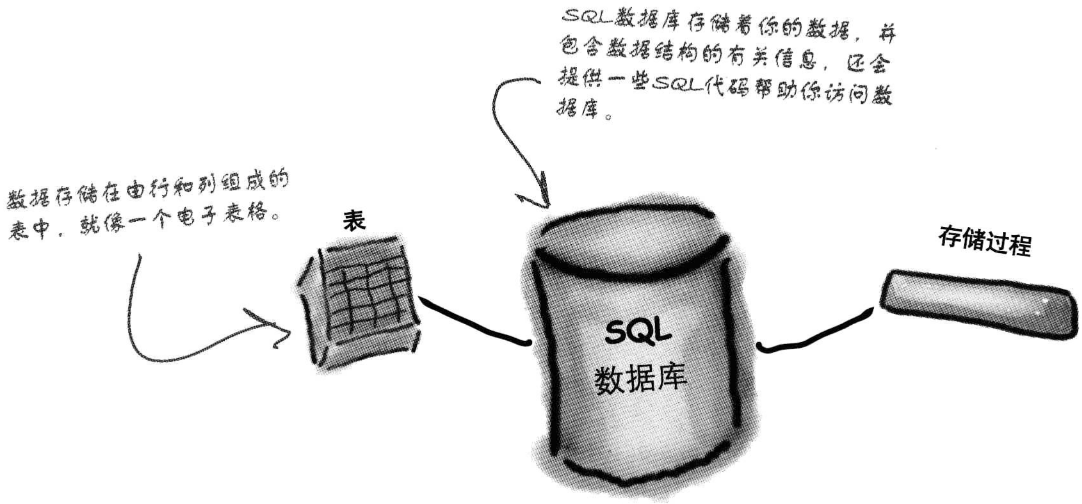
Visual Studio 2010 Professional 和Team Foundation版本没有Database Explorer窗口。实际上，这两个版本会提供一个Server Explorer窗口，Database Explorer所能做到的这个窗口都能做到，此外还允许访问网络上的数据。

# IDE创建一个数据库



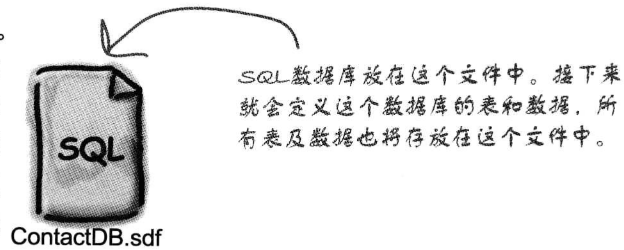

告诉IDE向工程增加一个新的SQL数据库时，IDE会为你创建一个新数据库。SQL数据库系统采用一种有组织、相互关联的方式存储数据。IDE已经提供了维护数据和数据库所需的所有工具。

SQL数据库中的数据存储在表中。现在可以把表认为是一个电子表格。它把信息按行和列组织。列就是数据类别，比如联系人的名字和电话号码；各行分别是一张联系卡片的数据。




# SQL是数据库自己的语言

SQL代表结构化查询语言（Structured Query Language）。这是一种在数据库中访问数据的编程语言。它有自己的语法、关键字和结构。SQL代码采用语句(statement)和查询(query)的形式，用来访问和获取数据。SQL数据库可以有存储过程，存储过程(strored procedure)就是一组SQL语句和查询，存储在数据库中，可以在任何时间运行。IDE会自动为你生成SQL语句和存储过程，使你的程序能够访问数据库中的数据。

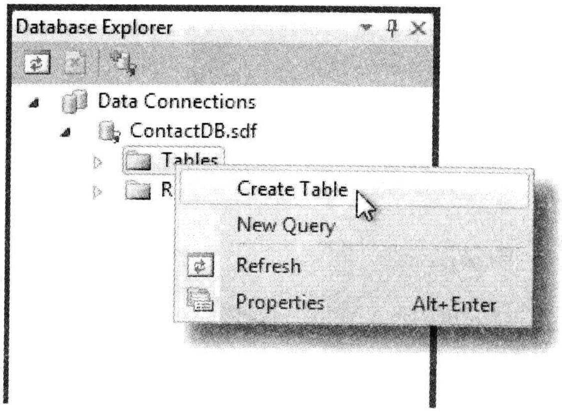



[来自市场的声音：可以在这里插放一个"Head First SQL"的广告吗?]

# 为Contact List应用创建数据库表

现在有了一个数据库，接下来需要在其中存储信息。不过，实际上信息必须放在数据库表中，数据库使用这种数据结构（表）来保存所有数据。下面为这个应用创建一个名为“People”的表来存储所有联系信息：

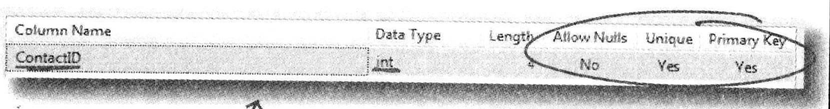
1. **为ContactDB数据库增加一个表。**
在Database Explorer中鼠标右键单击Tables，并选择Add New Table。这会打开一个窗口，可以在其中为刚创建的表定义列。




现在需要向这个表增加列。首先，向这个新People表增加一个名为ContactID的列，使每个联系记录都有唯一的ID。

2. **向People表增加一个ContactID列。**
在列名（Column Name）字段中输入“ContactID”，并从数据类型（Data Type）下拉框中选择Int。切记Allow Nulls要设置为No。

最后，将这一列置为这个表的主键。选中刚创建的ContactID列，再单击主键按钮。这就告诉数据库每个记录都有一个唯一的主键。




增加一个新列，名为“ContactID”，数据类型为“int”。“Allow Nulls”一定要设置为No，“Unique”要设置为Yes，Primary Key要设置为“Yes”。

## there are no Dumb Questions

**问：** 虽然你前面提到过，不过还想再问一次，什么是列？

**答：** 列就是表中的一个字段。所以，在People表中，可能有一个FirstName列和一个LastName列。列还总有一个数据类型，如String、Date或Bool。

**问：** 为什么需要这个ContactID列？

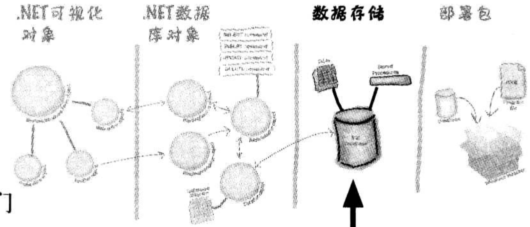
**答：** 在大多数数据库表中，这有助于保证各个记录都有一个唯一的ID。因为我们要存储个人的联系信息，所以我们决定为此创建一个列，并命名为ContactID。

**问：** 数据类型int是什么意思？

**答：** 数据类型会告诉数据库某个列需要存放哪种类型的信息。int代表integer，也就是一个整数。所以ContactID列中会存放整数。

**问：** 内容太多了。这些我都得掌握吗？

**答：** 不必。如果你现在还不能完全理解所有内容，也没有关系。现在的重点是熟悉使用Visual Studio IDE建立窗体并运行程序的基本内容（如果你确实现在就希望更多地了解数据库，可以另外参考《Head First SQL》）。




你现在的位置

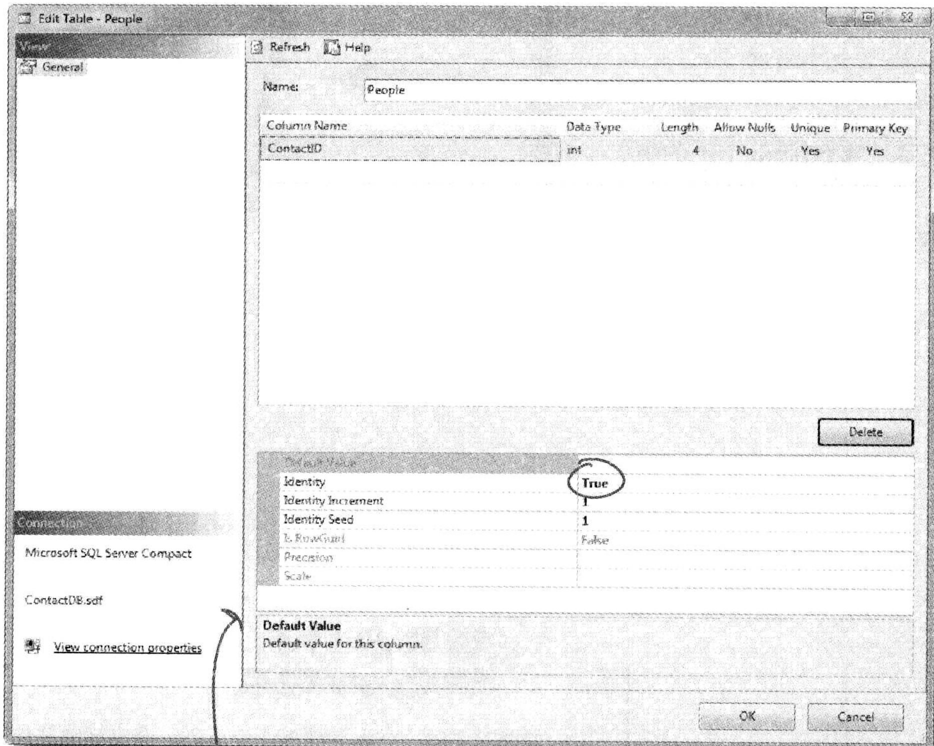
❸ **告诉数据库自动生成ID。**

由于ContactID是面向数据库的一个数，而不是面向我们的用户，所以可以告诉数据库自动地处理这些ID的创建和赋值。这样一来，我们就不必操心为此编写任何代码。

在表下面显示的属性中，将Identity设置为“True”，使ContactID成为表的一个标识列。

另外，确保在窗口上方的Name框中指定表名“People”。

要用这个窗口来定义数据库表以及其中存储的数据。



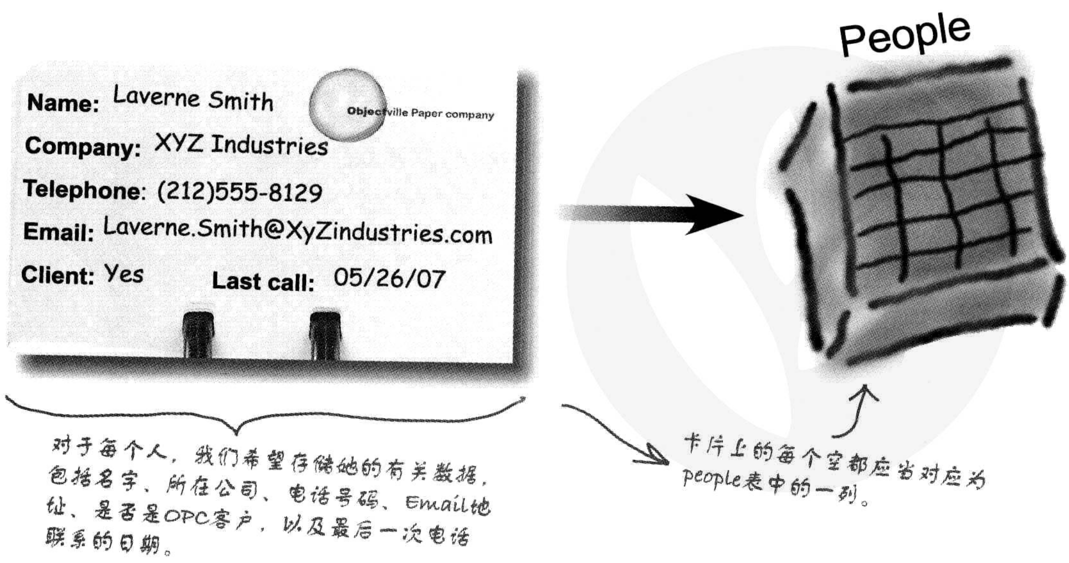

主键可以帮助你的数据库快速地查找记录。由于主键是程序查找记录的主要途径，所以这个列总要有一个值。

这样一来，只要增加新记录，就会自动更新ContactID字段。

需要单击右列，并从Identity旁边的下拉框选择“True”，从而指定ContactID作为这个表的记录标识列。

# 联系卡片上的空都是People表中的列

既然已经为这个表创建了主键，就需要定义将在数据库中维护的所有字段。对应手写联系卡片上的各个数据域，应当在People表中创建一个相应的列。




**BRAIN POWER**

如果对应同一个人存储多个记录行，会出现哪些问题？

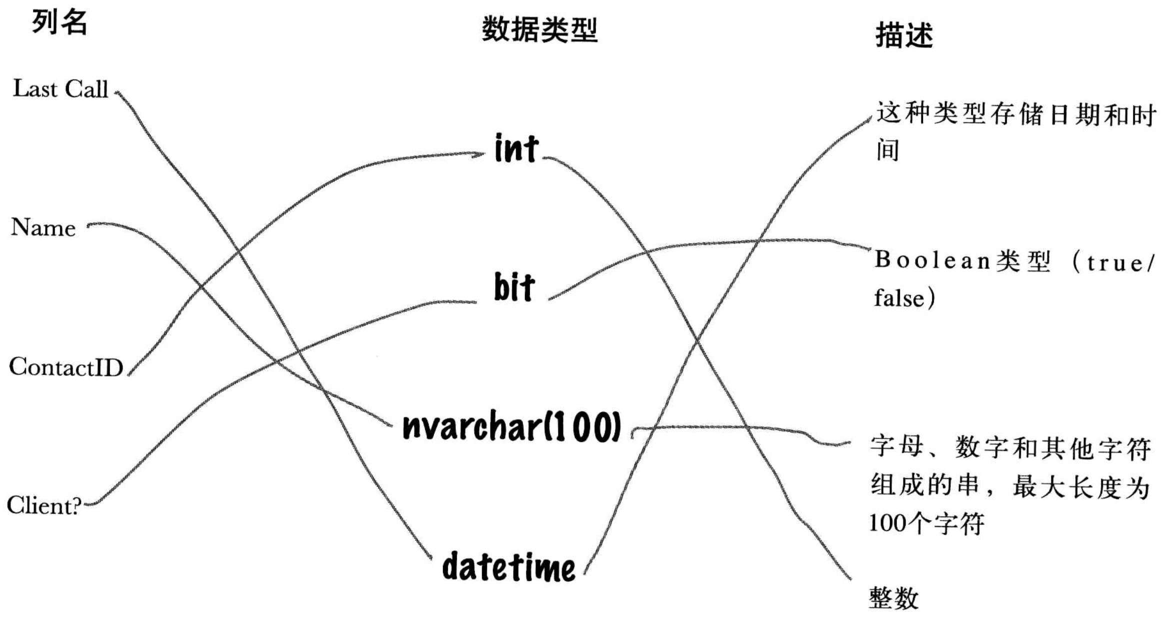
## WHO DOES WHAT?

既然已经创建了People表和一个主键列，就需要为所有数据域增加相应的列。看看你能不能搞清楚表中各个列应当选用哪种数据类型，并将数据类型与正确的描述配对。

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| Last Call | int | 这种类型存储日期和时间 |
| Name | bit | Boolean类型（true/false） |
| ContactID | nvarchar(100) | 字母、数字和其他字符组成的串，最大长度为100个字符 |
| Client? | datetime | 整数 |

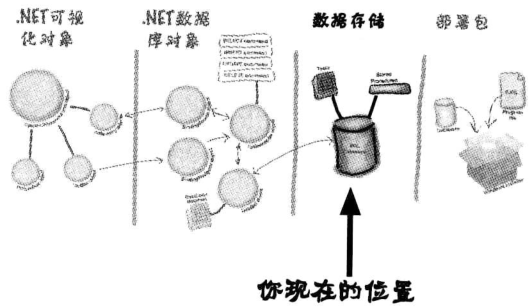
## WHO DOES WHAT?

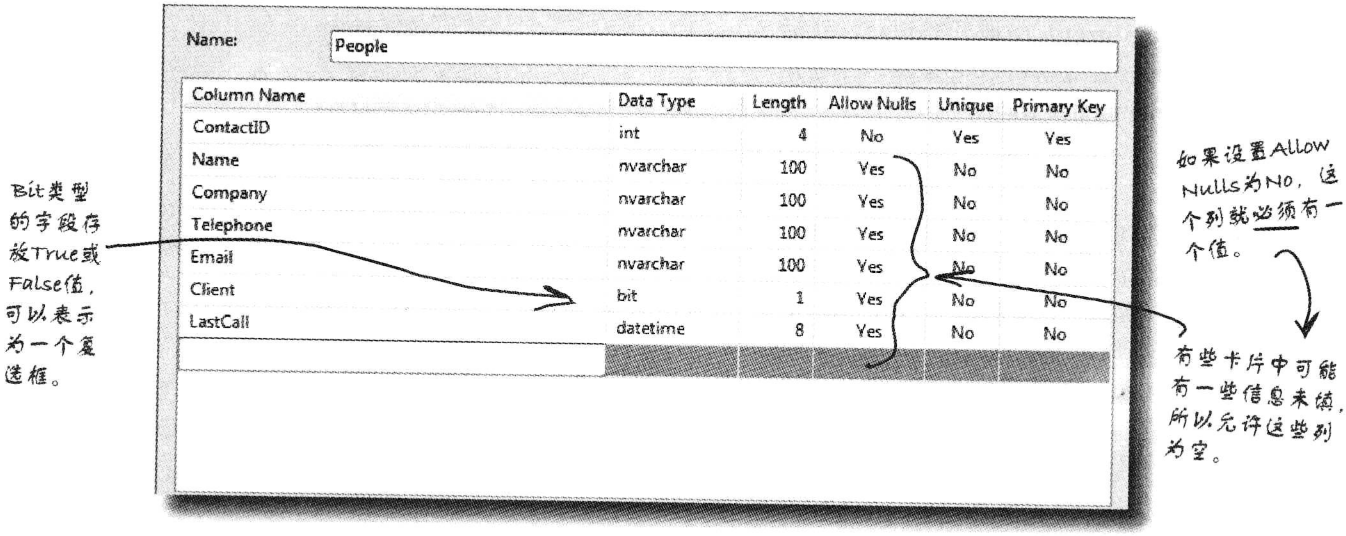
既然已经创建了People表和一个主键列，就需要为所有数据域增加相应的列。看看你能不能搞清楚表中各个列应当选用哪种数据类型，并将数据类型与正确的描述配对。



# 完成表的构建

回到输入ContactID列的窗口，再增加联系卡片上的另外5个列。完成后，这个数据库表将如下所示：



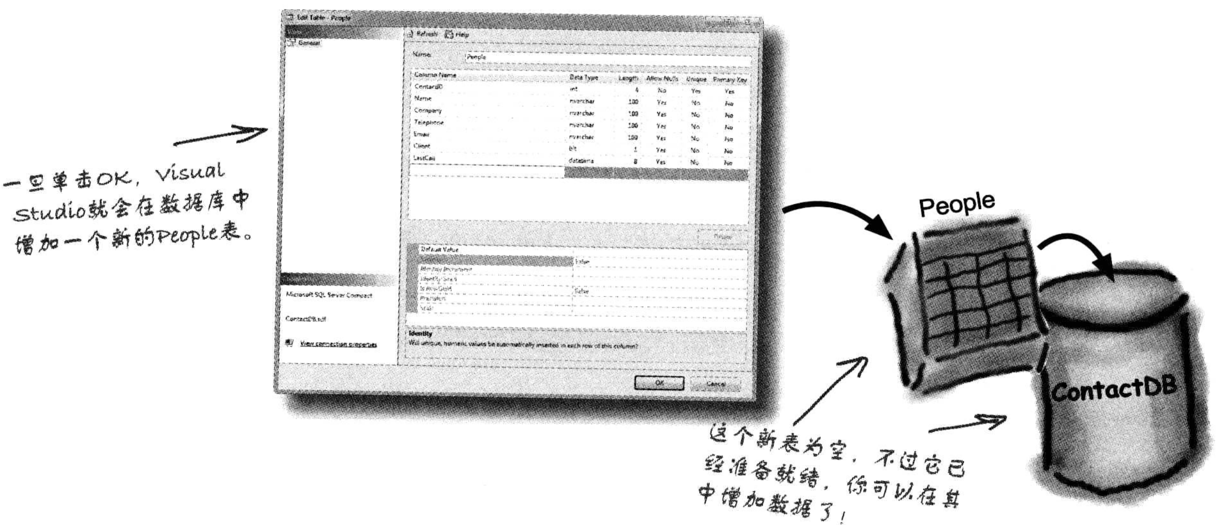



Name: People

| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key |
|---|---|---|---|---|---|
| ContactID | int | 4 | No | Yes | Yes |
| Name | nvarchar | 100 | Yes | No | No |
| Company | nvarchar | 100 | Yes | No | No |
| Telephone | nvarchar | 100 | Yes | No | No |
| Email | nvarchar | 100 | Yes | No | No |
| Client | bit | 1 | Yes | No | No |
| LastCall | datetime | 8 | Yes | No | No |

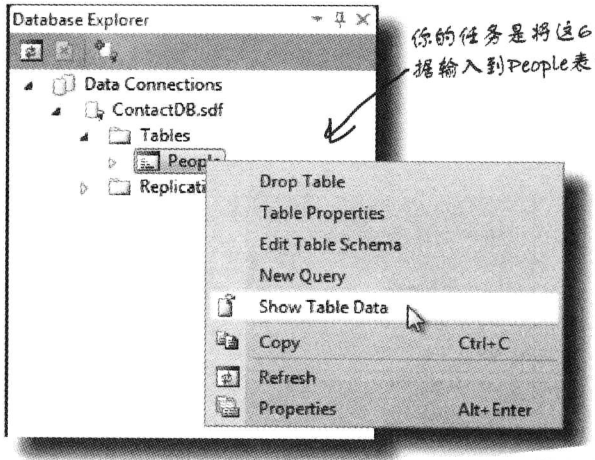


单击OK按钮保存这个新表。这会在你的数据库中增加一个空表。



# 将卡片数据插入到数据库

现在准备把卡片输入到数据库中。以下是老板的部分联系卡片，我们将用这些卡片在数据库中建立几条记录。

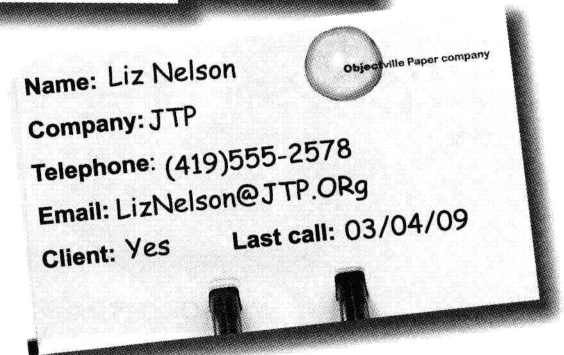


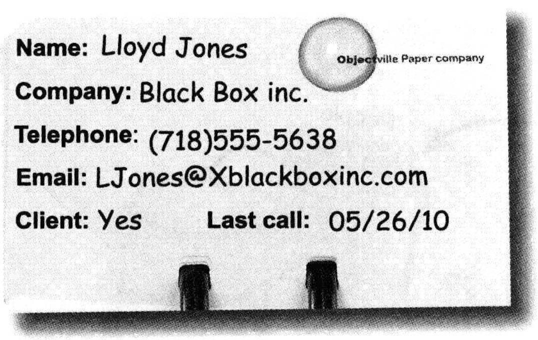

你的任务是将这6个卡片的数据输入到People表中。

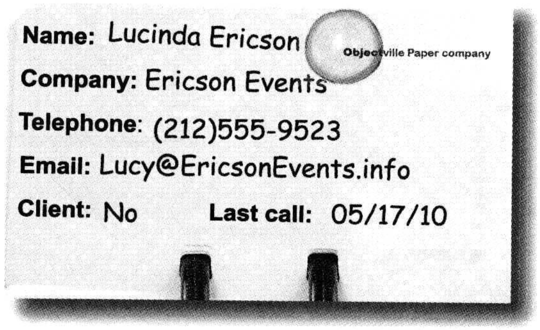
1. 在Database Explorer（或Server Explorer）中展开Tables，然后鼠标右键单击People表，并选择Show Table Data。

2. 看到主窗口中出现表格后，继续增加下面的所有数据（刚开始你会看到所有值都为NULL，增加第一行时只需在这些NULL值上直接输入。不用管数据旁出现的省略号）。不需要填ContactID列，这一列会自动填充。

Client列中要键入“True”或“False”。SQL采用这种方式存储“Yes”或“No”信息。






Objectville纸业公司地处美国，所以CEO采用这种方式写日期，05/26/10就表是2010年5月26日。如果你的机器设置为一个不同的位置，可能需要采用不同方式输入日期，也许需要使用26/05/10的形式。

③ 一旦输入了所有这6条记录，再从File菜单选择Save All，就会把这些记录保存到数据库。

"Save All" 告诉IDE要保存应用中的所有内容。这与 "Save" 有所不同，"Save" 只是保存你正在处理的那个文件。

## there are no Dumb Questions

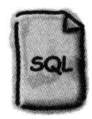
**问：** **数据输入之后会怎么样呢？它们到哪儿去了？**

**答：** IDE会自动地把你输入的数据存储到数据库中的People表。这个表、表中的列、数据类型以及表中的所有数据都将存储在SQL Server Compact数据库文件中（ContactDB.sdf）。这个文件将作为工程的一部分保存，IDE会更新这个文件，就像你对代码做了修改后IDE会更新代码文件一样。

**问：** **好了，我已经录入了这6条记录。它们将永远作为这个程序的一部分吗？**

**答：** 不错，就像你编写的代码和构建的窗体一样，这些记录也将成为程序的一部分。不过有一点区别，这些记录并不编译到一个可执行程序中，而是会复制ContactDB.sdf文件并与这个可执行文件一同存储。应用需要访问数据时，会读/写程序输出目录中的ContactDB.sdf。

这个文件实际上是一个SQL数据库，你的程序可以利用IDE生成的代码使用这个数据库。




ContactDB.sdf

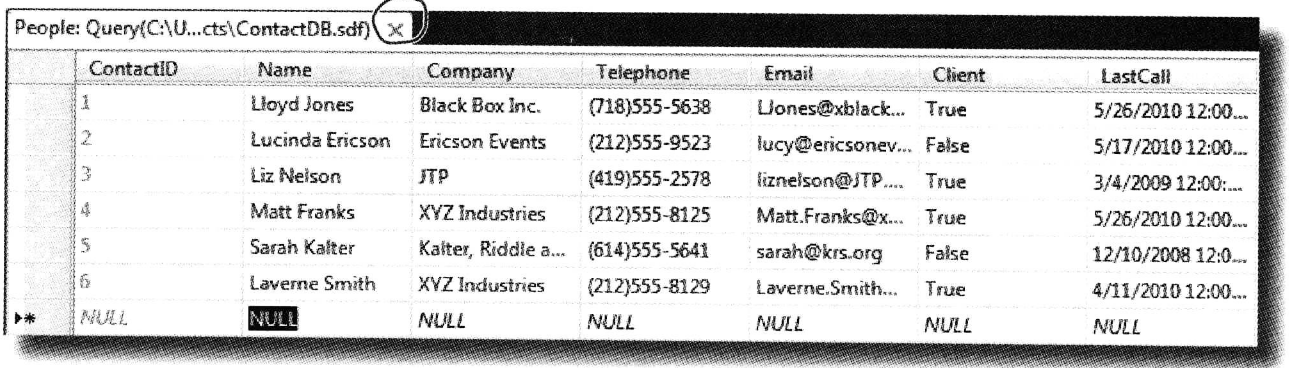
# 利用一个数据源连接窗体与数据库对象

终于该构建.NET数据库对象了，窗体将使用这些对象与数据库通信。我们需要一个数据源(data source)，这实际上是一组SQL语句，程序将使用这些语句与ContactDB数据库通信。

一旦数据输入完毕，需要关闭数据录入窗口才能回到窗体。

**❶ 再回到应用的窗体。**

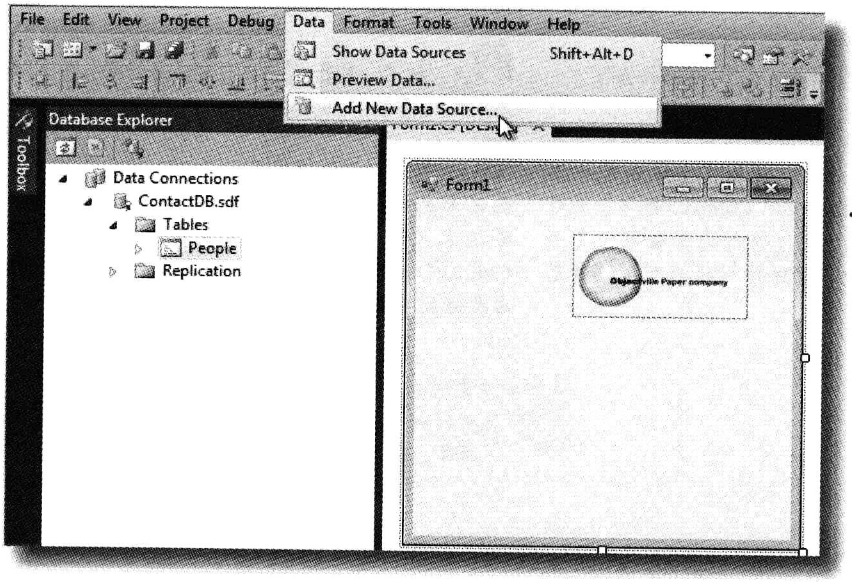
关闭People表和ContactDB数据库图表。现在应该能看到Form1.cs [Design]页。



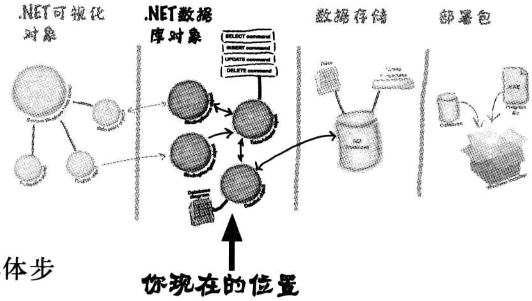

| ContactID | Name | Company | Telephone | Email | Client | LastCall |
|---|---|---|---|---|---|---|
| 1 | Lloyd Jones | Black Box Inc. | (718)555-5638 | LJones@xblack... | True | 5/26/2010 12:00... |
| 2 | Lucinda Ericson | Ericson Events | (212)555-9523 | lucy@ericsonev... | False | 5/17/2010 12:00... |
| 3 | Liz Nelson | JTP | (419)555-2578 | liznelson@JTP.... | True | 3/4/2009 12:00:... |
| 4 | Matt Franks | XYZ Industries | (212)555-8125 | Matt.Franks@x... | True | 5/26/2010 12:00... |
| 5 | Sarah Kalter | Kalter, Riddle a... | (614)555-5641 | sarah@krs.org | False | 12/10/2008 12:0... |
| 6 | Laverne Smith | XYZ Industries | (212)555-8129 | Laverne.Smith... | True | 4/11/2010 12:00... |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

**❷ 向应用增加一个新数据源。**

现在这应该很简单了。单击Data菜单，然后从下拉菜单选择“Add New Data Source……”。




所创建的数据源将处理窗体与数据库之间的所有交互。




❸ **配置新数据源。**

现在需要设置这个数据源来使用ContactDB数据库。具体步骤如下。

★ 第1步：选择一个数据源类型。选择Database，并单击Next按钮。

★ 第2步：选择一个数据库模型。选择Dataset并单击Next按钮。

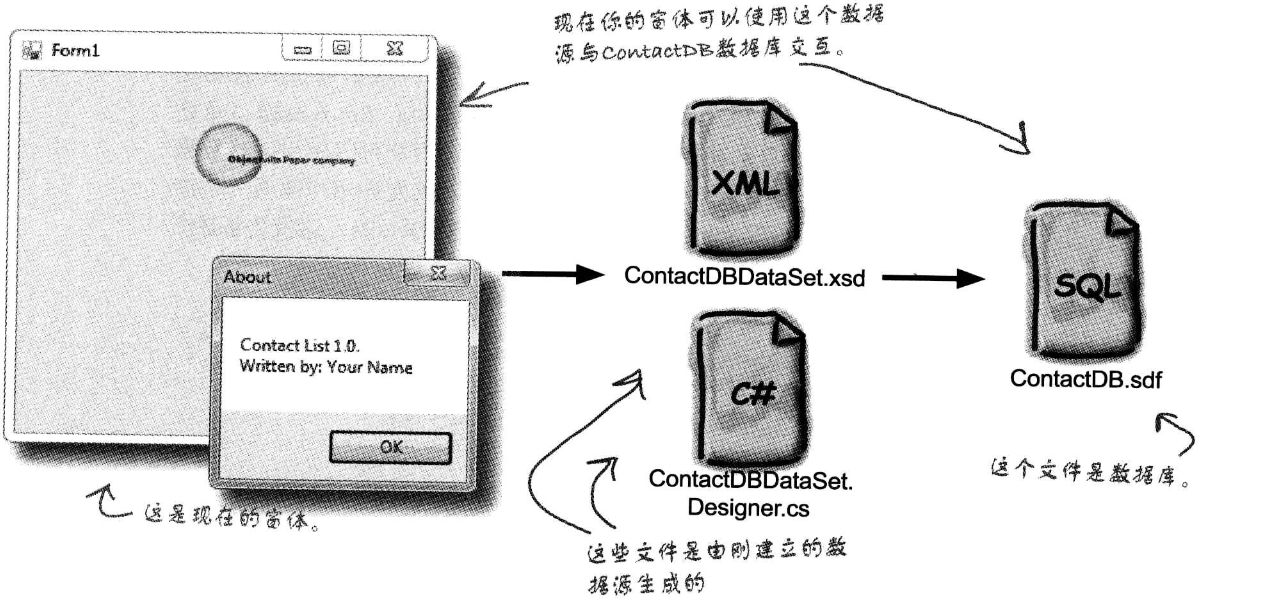
★ 第3步：选择数据连接。应该可以在下拉框中看到你的Contact数据库。单击Next。

★ 第4步：选择数据库对象。单击Table复选框。

★ 确保Dataset Name域中值为“ContactDBDataSet”，并单击Finish。

这几步将把这个新数据源与ContactDB数据库中的People表连接。

在非Express版本中，可能问你是否将连接保存在应用配置中，回答“Yes”。



# 为窗体增加数据库驱动控件

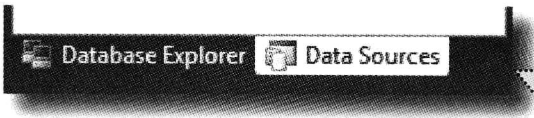
现在再回到我们的窗体来增加更多控件。不过这些不只是一般的控件，它们会绑定到我们的数据库以及People表中的列。这说明，在窗体上对其中某个控件的数据做了修改之后，数据库中相应列中的数据也会自动修改。

这要花点工夫，不过现在先创建与数据存储交互的窗体对象。

下面来介绍如何一些创建数据库驱动控件：

**1 选择你想使用的数据源。**
从Data下拉菜单选择Show Data Sources。这会弹出Data Sources窗口，显示你为应用建立的所有数据源。

如果没有看到这个标签页，可以从Data菜单选择“Show Data Sources”。




还可以找到Database Explorer窗口下方的Data Sources页并点击。

这个窗口会显示所有数据源。这里我们只建立了一个数据源，不过你完全可以为不同的表或数据库建立更多数据源。

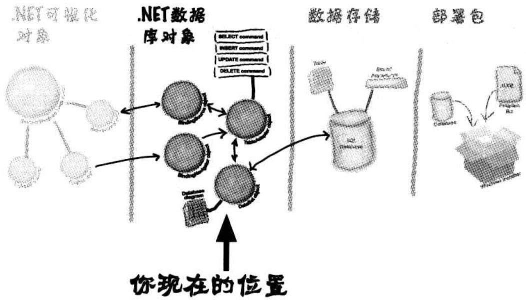
**2 选择People表。**
在ContactDBDataSet下面应当能看到People表和其中的所有列。单击People表旁边的“expand”将它展开，你会看到为这个表增加的所有列。在Data Sources窗口单击People表，把它拖到窗体上，IDE会自动在窗体上增加数据控件，用户可以使用这些数据控件浏览和输入数据。默认地，IDE会增加一个DataGridView，这就允许用户使用一个庞大的电子表格式控件处理数据。单击People表旁边的箭头，并选择Details。这就告诉IDE在窗体中为表中各个列增加单个的控件。




已创建的全部列都会显示在这里。

单击这个小箭头，并选择Details告诉IDE为窗体增加单个控件，而不是增加一个庞大的电子表格式数据控件。

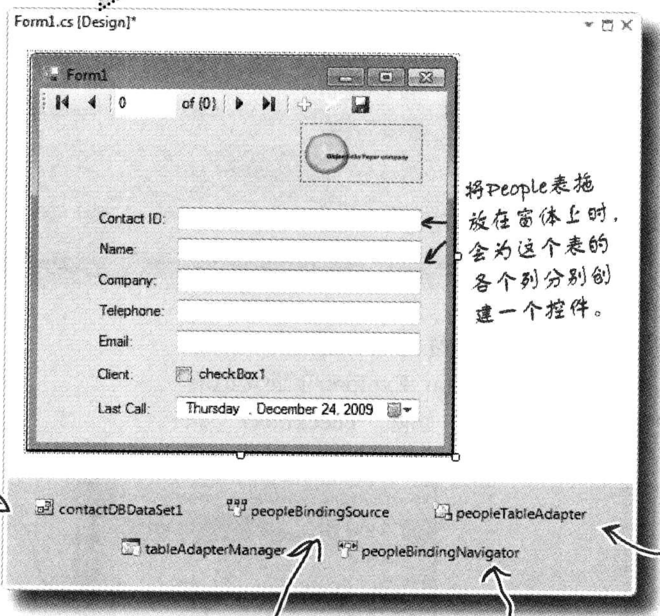
如果在IDE中打开窗体设计窗口，就会看到这个下拉框。它允许你直接将数据控件从数据源拖到窗体上。




❸ **创建绑定到People表的控件。**

把People表拖放到窗体设计窗口中的窗体上。应当能看到对应数据库中的各个列会出现一些控件。现在先不必太在意它们的外观，只要确保这些控件都在窗体上出现。

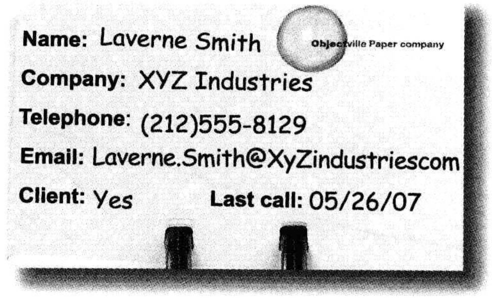
> 如果不小心点到正在处理的这个窗体的外面，完全可以通过单击“Form1.cs [Design]”页或者从Solution Explorer打开 Form1.cs再回到这个窗体。



# 好程序使用起来应当简单明了

目前，这个窗体已经能工作了。不过看起来不是太好。你的应用应当不只是提供功能而已，还应该易于使用。只需简单的几步，就可以让窗体看起来更像本章最前面使用的纸质卡片。

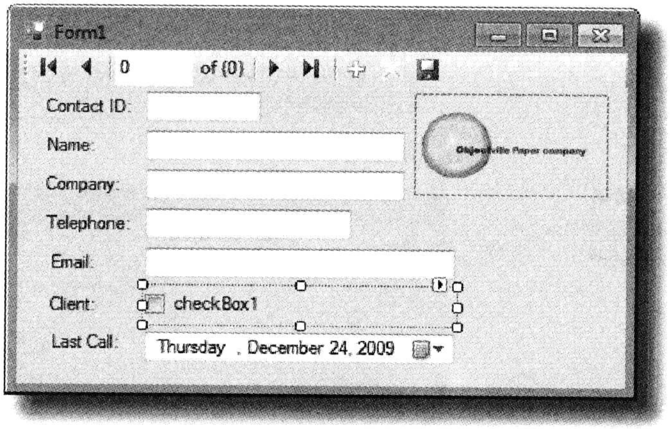
如果窗体看上去很像联系卡片，将更加直观。




① **摆放数据域和标签。**

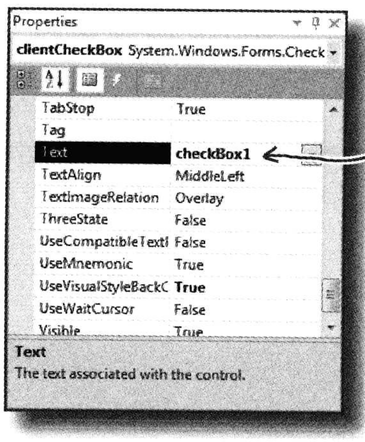
沿窗体左边摆放数据域和标签。这样一来，你的窗体看上去就会类似于其他应用，让用户使用时更舒服一些。

拖动控件时窗体上会显示蓝线。这些蓝线有助于将数据域摆放整齐。

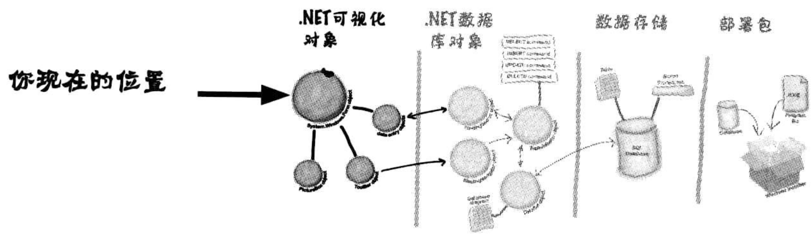



② **修改Client复选框的Text属性。**

第一次把字段拖到窗体上时，Client复选框右边会有一个标签，需要把这个标签删除。在Solution Explorer下面可以看到属性窗口。向下滚动到Text属性，删除“checkbox1”标签。




删除这个词，会去掉标签。



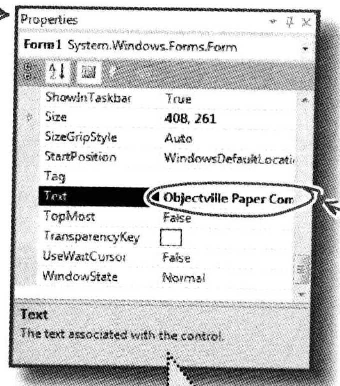

**③ 让应用看起来更专业。**

可以改变窗体名，为此单击窗体中任何空白位置，在IDE的Properties窗口中查找Text属性。将窗体名改为“Objectville Paper Company Contact List”。

还可以在这个窗口中去掉最大化和最小化按钮，首先找到MaximizeBox和MinimizeBox属性，再将这两个属性值都设置为False。

之所以要去掉最大化按钮，原因在于，将窗体最大化并不会改变控件的位置，这样看起来会很怪异。

Properties窗口应该就在Solution Explorer下面，在IDE右下方的面板中。




Text属性控制着窗体标题栏上显示的标题。

如果没有看到一个Properties窗口，则可以从View下拉菜单选择Properties来打开这个窗口。

**一个优秀的应用应当不仅能正常工作，还要易于使用。要确保它提供一般用户所期望的行为，这绝对是个好主意。**

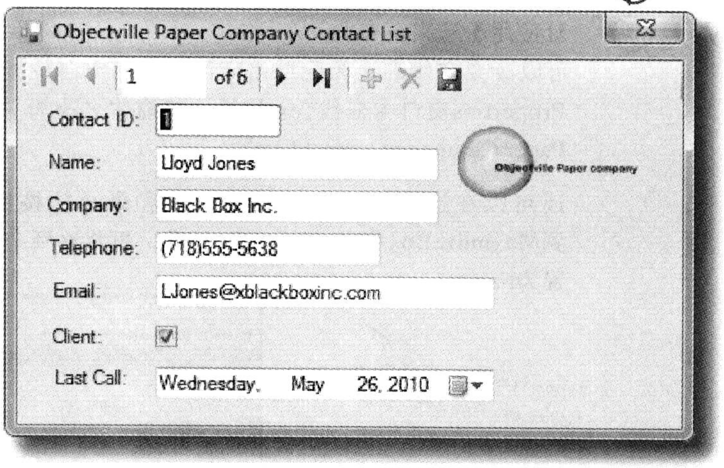
## 测试

好了，只剩下一件事了…… 运行这个程序，确保它能如你期望地工作！就像以前一样，按键盘上的F5键，或者单击工具条上的绿箭头按钮 ▶（或者，还可以从Debug菜单选择“Run”）。

可以在任何时刻运行你的程序，甚至在尚未完成的情况下也能尝试运行，不过如果代码中存在错误，IDE会指出错误并中止执行。

单击角上的X框，中止程序，转向下一步。

这些控件允许你分页查看数据库中的不同记录。




**构建应用会覆盖数据库中的数据。**

下一章还会更详细地讨论这个内容。

### IDE先构建再运行。

在IDE中运行程序时，实际上它会做两件事。首先，IDE会构建（build）程序，然后再执行（execute）程序。这包括几部分。首先编译（compile）代码，或者将代码转换为一个可执行文件。然后把已编译代码（连同有关的资源和其他文件）放在bin文件夹下的一个子目录中。

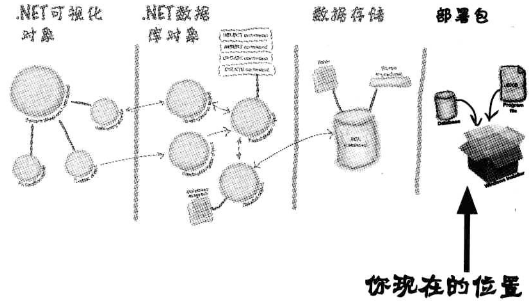
在这里，可以看到可执行文件和SQL数据库文件都放在bin/debug下。由于每次都会复制数据库，下一次在IDE中运行时，原先做的修改都会丢失。不过，如果从Windows运行可执行文件，你的数据则会保留（直到再次构建应用）。再次构建应用时，IDE会用一个新副本（其中包含在Database Explorer中建立的数据）覆盖SQL数据库。




**每次构建程序时，IDE会在bin文件夹下放置一个全新的数据库副本。这会覆盖你原来运行程序时增加的所有数据。**

调试程序时，如果代码有变化，IDE会重新构建程序。这意味着在IDE中运行程序时，你的数据库有时会被覆盖。如果直接从`bin/debug`或`bin/release`文件夹运行程序，或者如果你使用安装程序把它安装在你的机器上，就不会遇到这个问题。

# 怎么把你的应用变成所有人的应用？

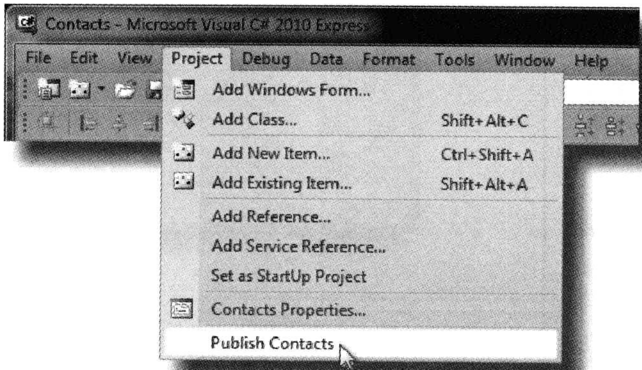



到目前为止，你已经有了一个不错的程序。但是它只能在你的机器上运行。这意味着，别人没办法使用这个应用，当然也不可能掏钱来买，你也无法证明你有多出色让别人来聘用你……你的老板和客户也无法看到你由数据库生成的报告。

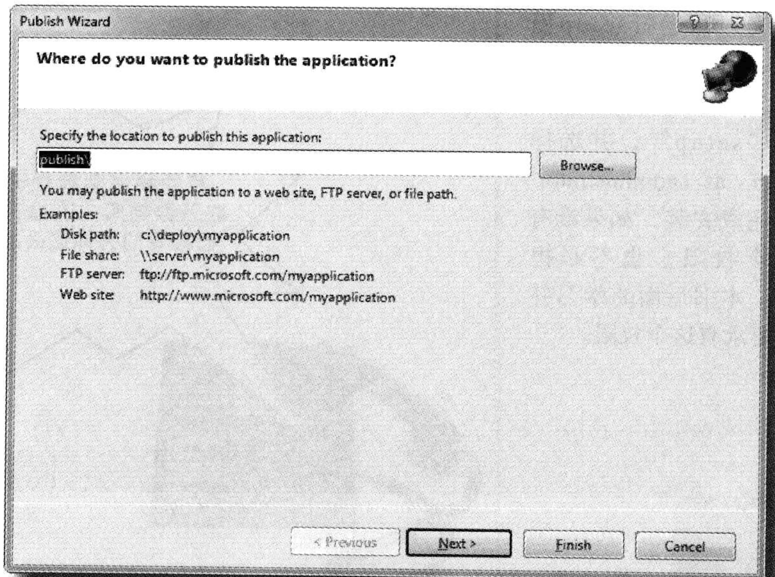
使用C#时，部署一个已创建的应用会非常容易。部署是指将一个应用安装到其他机器上。利用Visual C# IDE，只需两步就可以完成部署。

构建解决方案就是把文件复制到你的本地机器上。通过发布，将创建一个setup可执行文件和一个配置文件，从而可以在任何机器上安装你的程序。

**1** 从Project菜单选择Publish Contacts。



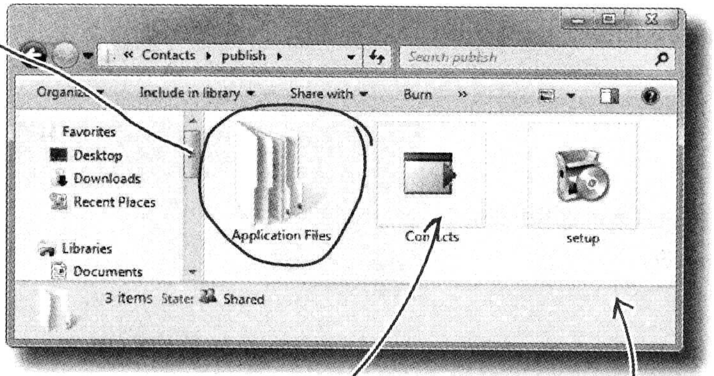

**2** 只需单击Finish，接受Publish Wizard向导中的所有默认值。你会看到，这将完成应用的打包，然后显示一个文件夹，其中包含setup.exe。




如果你使用的是Visual Studio Express，就会看到“Publish”在Project菜单下，但在其他版本中，这个菜单项可能在Build菜单下。

# 向用户提供应用

一旦创建了部署包，就会有一个新的文件夹publish。这个文件夹中包含的几项内容都是为了完成安装。对于你的用户来说，最重要的就是安装程序setup，利用这个安装程序，用户就能在他们自己的计算机上安装你的程序。

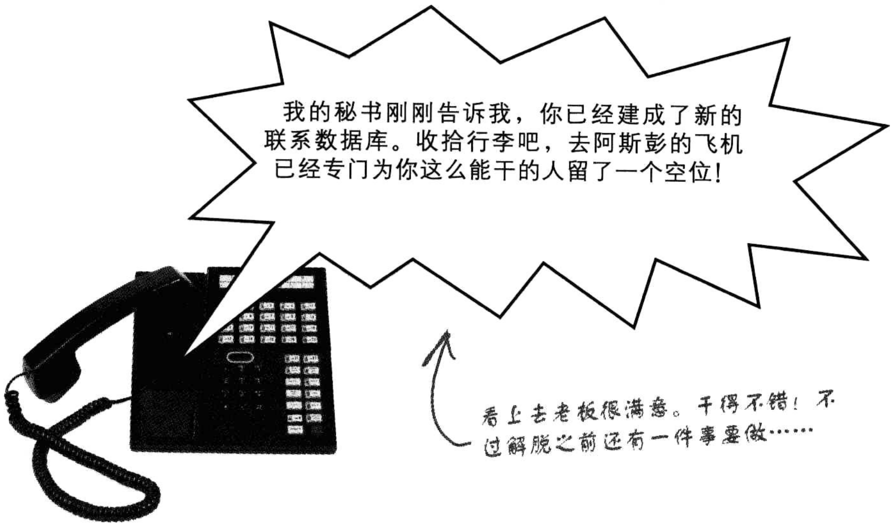




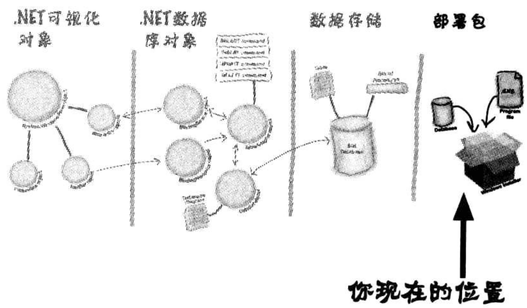


**你可能需要作为管理员才能运行安装程序。**

如果机器上还没有安装SQL Server Compact，安装程序会自动下载并安装SQL Server。在一些机器上，除非你作为管理员运行setup程序，否则这将无法做到。所以鼠标右键单击“setup”，并选择“Run as administrator”来完成安装。如果没有这个权限，也不必担心！本书后面的学习并不要求有这个权限。

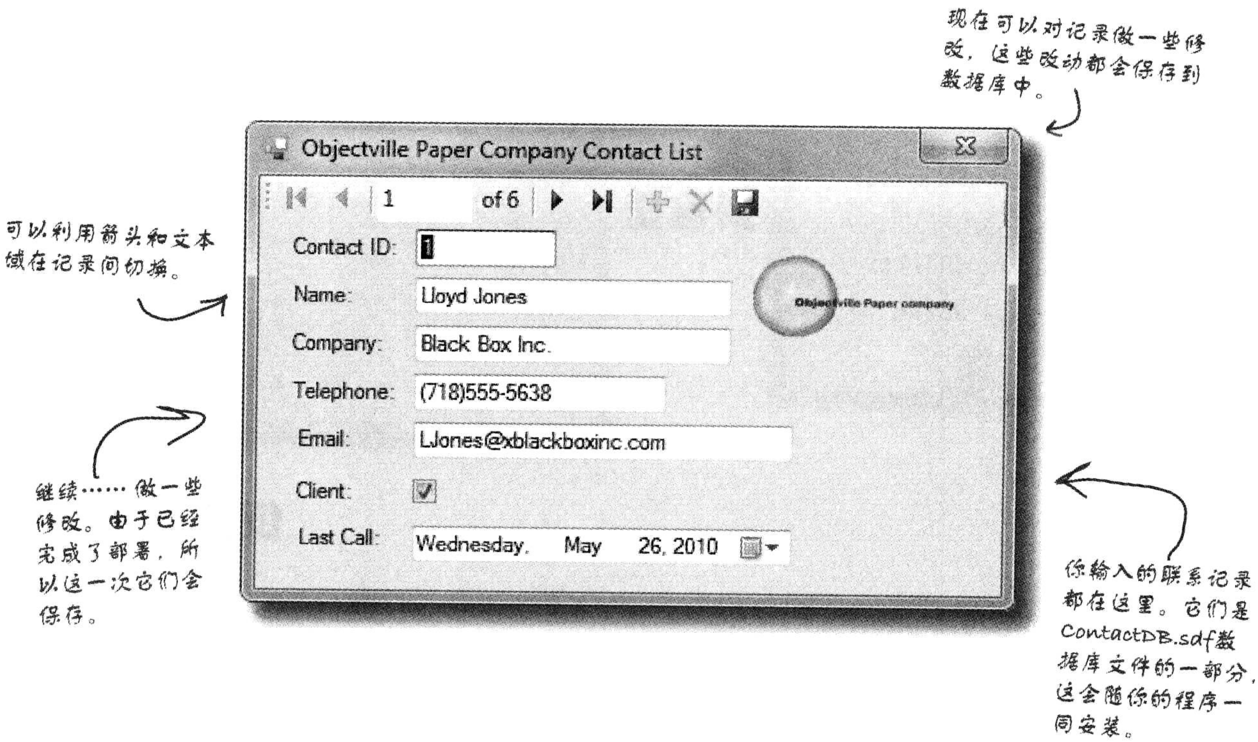


# 革命尚未成功：测试安装




开香槟庆祝之前，需要先测试部署和安装。如果一次都没有运行过，你是不会把程序交给别人的，对不对？

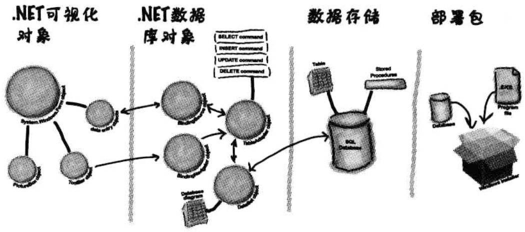
关闭Visual Studio IDE。单击安装程序，在你自己的计算机上选择一个位置来安装这个程序。现在从这个位置运行你的程序，确保它能如你所期望地正常工作。还可以增加和修改记录，这些都将保存到数据库中。



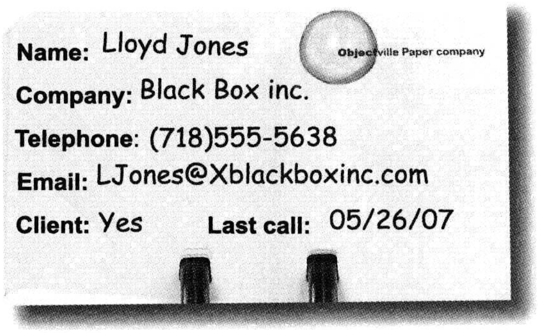

**一切都必须测试！要测试你的程序，测试你的部署，测试应用中的数据。**

## 你已经构建了一个完整的数据驱动应用

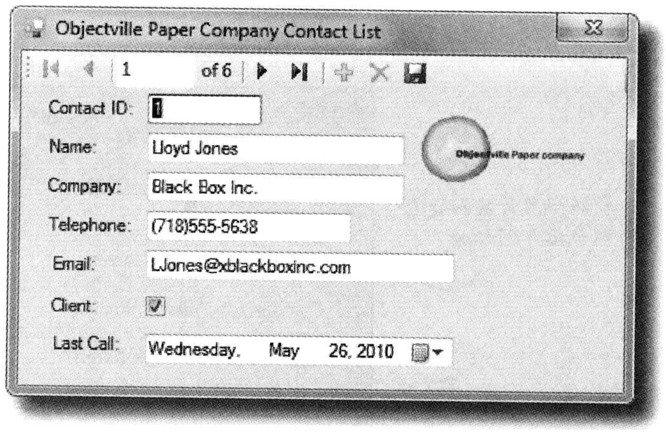
有了Visual Studio IDE，创建Windows应用、创建和设计数据库，以及将应用和数据库相关联都相当容易。甚至只需轻轻单击几次鼠标就能构建一个安装程序。




**从这里**




**到这里**

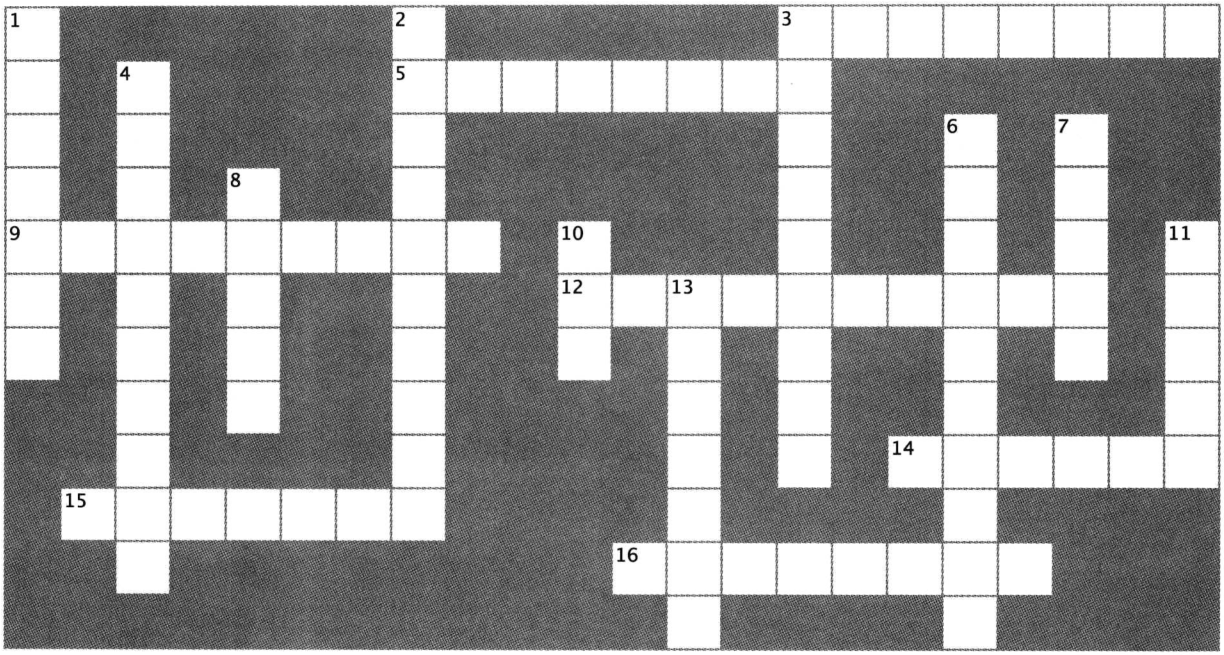



**仅片刻之间**

**Visual C#的强大之处在于，你可以非常迅速地建立和运行程序，然后集中精力考虑程序要实现的具体功能……而不必为大量窗口、按钮和SQL访问代码费心。**

# CSharp填字游戏

花些时间坐下来，用这个填字游戏测试你的C#词汇掌握得怎么样：所有答案都可以从这一章中找到。




## 横向

3.要在__________explorer中编辑SQL表的内容，并把它们绑定到你的程序。

5.这是一个图像、声音、图标或文件，采用一种便于对象轻松访问的方式关联到工程中。

9.要构建一个__________，从而能把你的程序部署到其他的计算机上。

12.IDE中的“I”代表什么。

14.双击一个控件时，IDE会为你创建__________，再由你为它增加代码。

15.数据库中的每一行包含很多__________，所有这些可能有不同的数据类型。

16.__________Explorer中会显示工程中的所有文件。

## 纵向

1.代码转换为可执行文件时完成的过程。

2.通过修改__________来改变窗体上控件的外观或行为。

4.ObjectvilLe纸业公司Contact List程序中的“About”框是一种__________。

6.用__________显示Objectville纸业公司的logo。

7.开始构建应用之前，一定要考虑用户以及用户的__________。

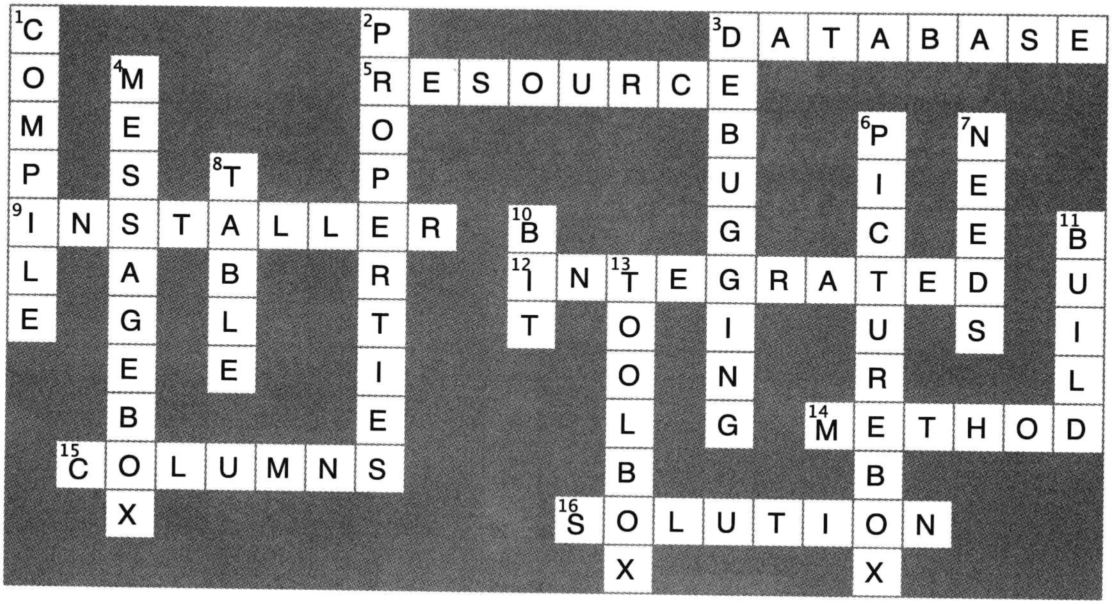
8.数据库可以使用很多__________来存储数据。

10.SQL数据库中可以使用__________数据类型存储true/false值。

11.运行程序前，IDE会做__________来创建可执行文件，并把文件移动到输出目录。

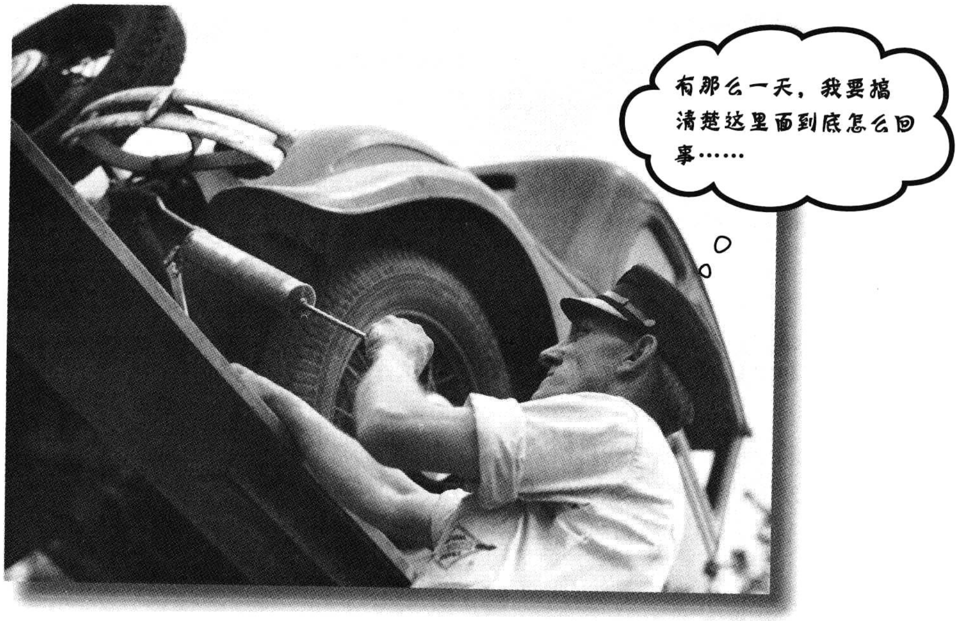
13. 把控件从__________拖到窗体上。

# CSharp填字游戏答案



# 2 都只是代码

# 幕后的故事




**你是一个程序员，不只是一个IDE用户。**

用IDE可以做很多事情。不过不能过于夸大IDE，它的作用总是有限的。当然，构建一个应用时要完成大量重复的任务，而且IDE最擅长为你做这些事情。不过使用IDE还只是一个开始。可以让你的程序做更多的事情，唯一的办法就是编写C#代码。一旦掌握了编写代码的窍门，那么没有什么能难得住你的程序。

# 当你这么做时……

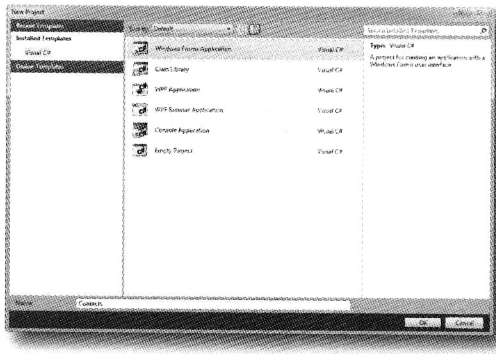
IDE是一个功能强大的工具，不过仅此而已，这只是一个可供使用的工具。每次在IDE中修改你的工程或者拖放某个控件时，它都会自动创建代码。IDE确实很擅长编写那些“样板”代码，也就是可以轻松重用而不需要太多定制的代码。

所有这些任务都需要利用标准动作和样板代码来完成。这些方面IDE最为擅长，也最有帮助。

下面来看在典型的应用开发中IDE会做哪些工作，当你……

**1 创建一个Windows Forms Application工程。**

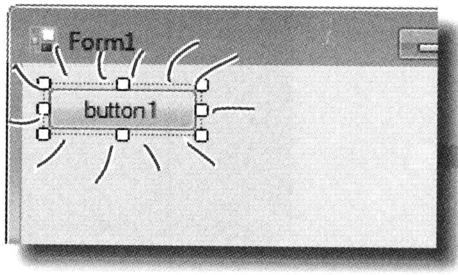
IDE允许构建多种不同类型的应用，不过我们现在强调的是Windows Form应用。所谓Windows Form应用，就是指包含可视化元素（如窗体和按钮）的应用。



要确保创建一个Windows Forms Application工程。这会告诉IDE创建一个空窗体，并把它增加到你的新工程中。

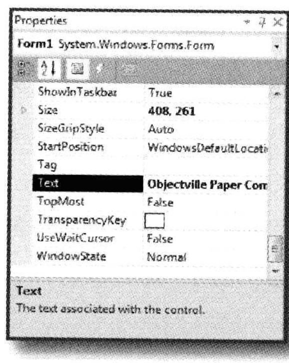
**2 把一个按钮从工具箱拖到窗体上，然后双击这个按钮。**

可以借助按钮在窗体中做一些事情。我们将使用大量按钮来详细分析C#语言的各方面内容。在你编写的几乎每一个C#应用中，按钮都将是不可或缺的一部分。




**3 在窗体上设置属性。**

IDE中的Properties窗口是一个非常强大的工具，可以用来改变程序中所有对象的属性，包括窗体上控件的所有可视化属性和功能属性，数据库的属性，甚至工程本身的选项。




IDE中的Properties窗口只是一种自动编辑Form1.Designer.cs中特定代码块的简便方法。如果不利用这个工具而手工编辑这些代码，花费的时间会多得多。如果Properties窗口关闭，可以使用快捷键F4将它打开。

# ……IDE会这么做

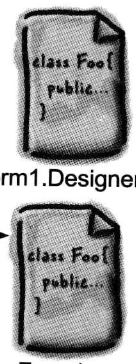
每次在IDE中做一个修改时，IDE就会对代码进行修改，这意味着它会修改包含该代码的文件。有时只是修改其中的几行代码，但有些情况下还可能向工程中增加完整的文件。

这些文件由一个预定义模板创建，其中包含创建和显示一个窗体的基本代码。

1 ……IDE会为这个工程创建文件和文件夹。




2 ……IDE为Form1.Designer.cs文件增加相应代码从而向窗体增加按钮，然后在Form1.cs文件中增加代码来处理按钮点击事件。




```
private void button1_Click(object sender, EventArgs e)
{

}
```

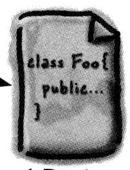
IDE知道如何增加一个空方法来处理按钮点击事件。不过它不知道要在其中放些什么——这是要由你完成的任务。

这个代码会增加到Form1.cs。

3 ……IDE打开Form1.Designer.cs文件，并更新一行代码。

IDE进入这个文件……

```
partial class Form1
{
   ⋮
       this.Text = "Objectville Paper Company Contact List";
   ⋮
}
```



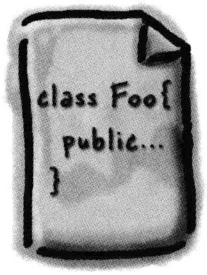

……并更新这行代码。

# 程序从哪里来

最初C#程序只是放在一组文件中的语句，不过最后却将作为一个程序在你的计算机中运行。下面介绍它从何而来。

## 每个程序最初只是一些源代码文件

你已经知道了如何编辑一个程序，也知道了IDE会把程序作为文件保存在一个文件夹中。这些文件就是你的程序，可以把它们复制到一个新的文件夹，打开后会看到所有内容：窗体、资源、代码，以及增加到工程中的所有一切。




可以把IDE认为是一种方便的文件编辑器。它会自动为你完成缩进，改变关键字的颜色，完成括号匹配，甚至还能建议下一个可能的词是什么。不过，归根结底，IDE所做的只是编辑构成程序的相关文件。

宏全可以用记事本(Notepad)构建你的程序，不过这样很费时间。

IDE将程序的所有文件打包到一个解决方案（solution）中，为此将创建一个解决方案(`.sln`)文件以及一个文件夹，这个文件夹中包含该程序的所有其他文件。解决方案文件包含解决方案中工程文件（扩展名为`.csproj`）的一个列表，这些工程文件则包含了与程序相关的所有其他文件的列表。在本书中，我们构建的解决方案只包含一个工程，不过可以很容易地使用IDE的Solution Explorer为你的解决方案增加其他工程。

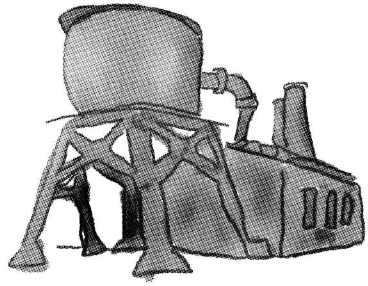
## .NET Framework提供了合适的工具来完成任务



C#只是一种语言，单凭它自己实际上什么也做不了。这就引入了.NET Framework。还记得Contacts窗体上去掉的最大化按钮吗？点击一个窗口的最大化按钮时，会有一些代码告诉这个窗口如何完成最大化来占据整个屏幕。这些代码就是.NET Framework的一部分。按钮、复选框、列表等都属于.NET框架，将窗体与数据库关联的内部控件也是.NET Framework的组成部分。将窗体关联到数据库的内部代码也属于.NET Framework。 .NET Framework提供了很多工具，可以用来绘制图形、读/写文件、管理集合等，对于程序员每天要完成的大量任务，所需的各种工具都可以在.NET Framework中找到。

.NET Framework中的工具划分为不同的命名空间(namespace)。之前已经见过这些命名空间（就在代码最上面的“using”代码行中）。其中包括一个名为System.Windows.Forms的命名空间，按钮、复选框和窗体就来自这里。只要创建一个新的Windows Forms Application工程，IDE就会增加必要的文件，使工程中包含一个窗体，而这些文件最前面都有一行“`using System.Windows.Forms;`”。

## 构建程序，创建一个可执行文件



在Build菜单中选择“Build Solution”时，IDE会编译(compile)你的程序。这是通过运行编译器(compiler)完成的，这个工具会读入程序的源代码，将代码转换为一个可执行文件(executable)。所谓可执行文件就是硬盘上一个扩展名为.exe的文件，双击这个文件就能运行你的程序。构建程序时，IDE将在该工程文件夹下的bin文件夹中创建这个可执行文件。发布你的解决方案时，将把这个可执行文件（以及所有其他必要的文件）复制到发布到的目标文件夹。

从Debug菜单选择“Start Debugging”时，IDE会编译程序，并运行可执行文件。它提供了一些相当高级的工具来调试(debugging)程序，调试就是指运行程序，而且能够暂停（或“中断”），以便你了解执行的情况。

## 程序在CLR内运行



双击可执行文件时，Windows将运行你的程序。不过在Window与你的程序之间还有另外的一“层”，称为通用语言运行时库(Common Language Runtime，CLR）。曾经有一段时间（就在不久以前，不过是在C#出现之前），编写程序没有这么容易，因为你必须处理硬件和底层设备的具体问题。你无法准确地知道别人会怎样配置他的计算机。CLR（通常称为一个虚拟机）会帮助解决所有这些问题，它会在你的程序和运行该程序的计算机之间完成某种“转换”。

现在你不必太担心CLR，知道有这样一个CLR，而且它会自动为你运行程序，这就足够了。随着后面内容的展开，你会对它有更多的了解。

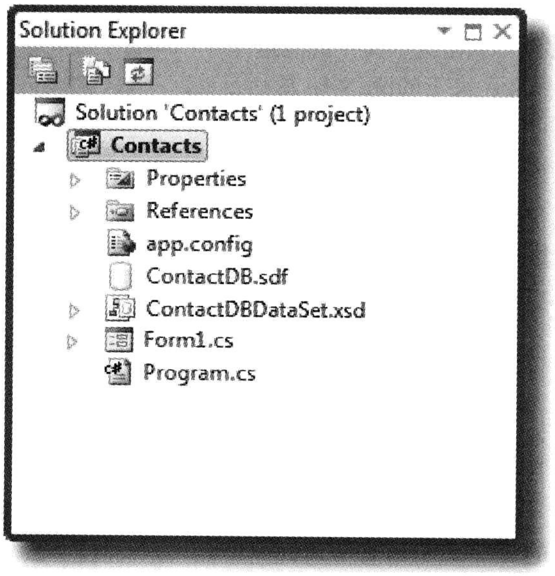
你会了解CLR为你做的所有工作。例如，它会很“节俭”地管理计算机的内存，发现你的程序用完某些数据时，就会帮你释放相关的内存。这些工作以前要由程序员自己来做，而现在你不必再为此费心。也许你当时不知道，但是CLR确实使C#的学习容易得多。

# IDE帮你编写代码

前面已经见识了IDE的部分功能。下面将更深入地分析IDE提供的一些工具。

**Solution Explorer会显示工程的所有内容。**

在类之间来回切换将花费你的很多时间，而完成这种切换最容易的方法就是使用Solution Explorer。以下是创建Objectville纸业公司Contact List程序之后的Solution Explorer。




Solution Explorer会显示解决方案文件夹中的不同文件。

**使用标签页在打开的文件间切换。**

由于程序往往分为多个文件，通常你会同时打开多个代码文件。如果是这样，每个文件会分别在代码编辑器中一个单独的标签页中显示。如果某个文件尚未保存，IDE会在该文件名旁边显示一个星号(*)。

这是窗体的资源文件，先前Objectville纸业公司的logo就增加到这个文件。

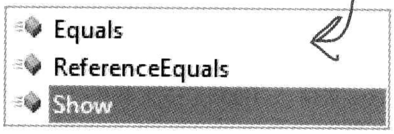



处理一个窗体时，可能同时会看到两个对应窗体的标签页，一个是窗体设计工具，另一个用于查看窗体的代码。按Ctrl+Tab可以快速在打开的窗口之间切换。

### IDE帮助你编写代码。

你注意过在IDE中输入代码时弹出的小窗口吗？这个特性称为智能提示（IntelliSense），这个特性非常有用。它的工作就是为你显示一些可能的选择，帮助你完成当前代码行。如果你输入MessageBox，然后输入一个点号（.），IDE就会知道完成这行代码有3种可行的做法：

```
MessageBox.
```

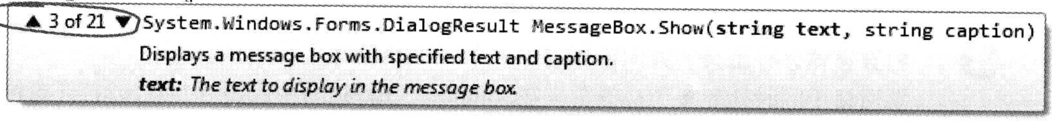



IDE知道MessageBox有3个方法，分别是Equals、ReferenceEquals和Show。如果你输入S，它就会选择Show。输入“（”或空格、Tab键或按Enter键，让IDE为你填入相应内容。如果需要输入大量相当长的方法名，这会大大节省你的时间。

如果选择Show，然后输入“(”，IDE的智能提示工具将显示完成这行代码的有关信息：

这说明，调用MessageBox的Show方法有21种不同的方式（就像显示不同按钮或图标时也是方法各异）。

```
MessageBox.Show(
```



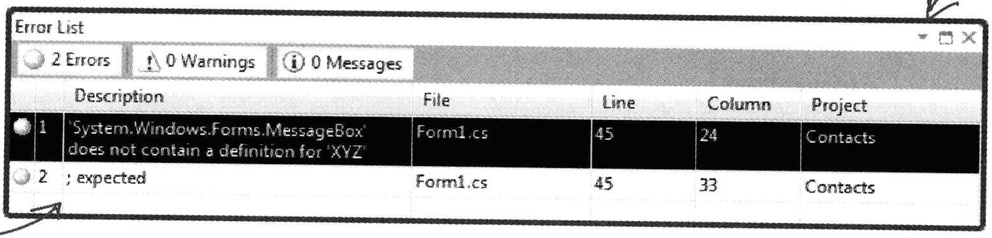

IDE还有一些快捷方法，称为片段（snippets），允许你输入一个缩写，由IDE填完其余的代码。以下就是一个有用的片段：输入mbox，然后按下两次Tab键，IDE就会为你填完MessageBox.Show方法。

```
MessageBox.Show("Test");
```

### Error List帮助你检查编译错误。

也许你还没有意识到C#程序中犯录入错误是多么容易，但很快你就会发现这一点！幸运的是，IDE提供了一个很棒的工具来排除这些错误。构建解决方案时，导致无法通过编译的所有问题都会显示在IDE下方的Error List窗口中：

使用Start Debugging在IDE中运行你的程序时，它做的第一件事就是构建程序。如果编译通过，程序就会运行，如果未能编译成功，则无法运行，将在Error List中显示错误。




语句最后缺少分号是导致程序无法顺利构建的最常见的原因之一。

双击一个错误，IDE就会跳至出问题的那行代码：

```
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.XYZ("hi")
}
```

IDE发现一个错误时会显示一个红色的下划线。

# 在IDE中做修改，同时也正在修改你的代码

看到“动手做！”标志时，请打开IDE，按照我们的步骤跟着做。我们会明确地告诉你做什么，并指出要注意的问题，这样能最大限度地利用我们给出的例子。

IDE对于为你编写可视化代码非常擅长。不过，不要光听我们讲。打开Visual Studio，创建一个新的Windows Form Application工程，亲身体验一下。

动手做！

1. **打开设计代码。**
在IDE中打开`Form1.Designer.cs`文件。不过这一次我们不是在窗体设计工具中打开，而是在Solution Explorer中鼠标右键单击这个文件并选择“View Code”来打开这个文件的代码。请找到Form1的类声明：

```
partial class Form1
```

注意到这是一个部分类了吗？稍后我们再介绍这个内容。

2. **打开窗体设计工具，并向窗体增加一个PictureBox。**
要习惯于同时处理多个标签页。在Solution Explorer中双击`Form1.cs`打开窗体设计工具。将一个新PictureBox拖到新窗体上。

3. **找到并展开设计工具为PictureBox控件生成的代码。**
然后回到IDE中的Form1.Designer.cs标签页。向下滚动，找到下面这行代码：

单击加号

```
⊞  Windows Form Designer generated code
```

单击这一行左边的+，展开代码。向下滚动，找到下面这几行：

```
//
// pictureBox1
//
this.pictureBox1.Location = new System.Drawing.Point(276, 28);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(100, 50);
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;
```

如果你的代码中对应Location和Size的行中数字与这里不同，也不必担心……

## 等等，请等一等!这是什么意思?

先把代码向上滚。看到了吗？在Windows Form Designer生成的代码最前面有以下代码。

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
```

大多数注释前面只有两个斜线(//)。但是IDE有时会增加这种有3个斜线的注释。

这些是XML注释，可以用这些注释建立代码的文档。在了解更多有关内容，可以翻到本书附录“其他”中的第1节。

对一个孩子来说，再没有比一个写着“不要碰”的大警示牌更具诱惑力的了。没错，你知道这里的“不要修改”确实让人跃跃欲试……那我们就动手改改吧，用代码编辑器来修改这个方法的内容！向窗体增加一个按钮，然后再做下面的工作：

1. **修改设置button1.Text属性的代码。你觉得这会对IDE中的Properties窗口有什么影响？**
   试试吧——看看会发生什么！现在回到窗体设计工具，查看Text属性。有变化吗？

2. **还是在窗体设计工具中，使用Properties窗口把Name属性改成其他值。**
   看看你能不能想办法让IDE改变Name属性。答案就在最上面的Properties窗口（“(Name)”下面）。代码有什么变化？代码中的注释呢？

3. **修改设置Location属性的代码，将值改为(0,0)，并修改Size属性使按钮相当大。**
   能起作用吗？

4. **回到设计工具，把按钮的BackColor属性改为其他颜色。**
   仔细查看Form1.Designer.cs代码。有没有增加代码行？

并不需要保存窗体或运行程序才能看到这些变化。只需在代码编辑器中做出修改，再单击设计工具标签页回到窗体设计工具，应该就能立即看到变化。

**通常，使用IDE来修改窗体中由设计工具自动生成的代码更为容易。不过，如果这样做，IDE中做的所有改变也将导致工程中的代码发生改变。**

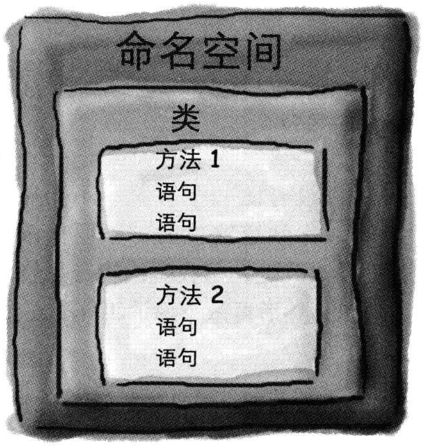
# 程序剖析

每个C#程序的代码结构都几乎完全一样。所有程序都使用了命名空间、类和方法，使代码更易于管理。

每次建立一个新程序时，要为它定义一个命名空间，使程序代码与.NET Framework类区别开。

类包含程序的一部分（不过，有些很小的程序可能只有一个类）。

类中有一个或多个方法。方法总是放在类中。方法由语句组成（如前面所看到的方法）。




## 下面更仔细地分析这些代码

打开Contact工程中`Form1.cs`的代码，下面一部分一部分来进行详细分析。

**1 C代码文件首先使用.NET Framework工具。**

可以看到，每个程序文件最前面都有一组`using`代码行。这些代码行告诉C#要使用.NET Framework中的哪些类。如果使用了其他命名空间中的其他类，那么还要为这些类增加相应的`using`代码行。由于窗体通常使用.NET Framework中很多不同的工具，所以IDE创建窗体并将窗体增加到工程时，会自动增加一组using语句。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

这些using代码行位于每个代码文件的最前面。它们告诉C#可以使用所有这些.NET Framework类。每个using代码行向程序指出：这个.cs文件中的类会使用一个特定.NET Framework命名空间中的所有类。

有一点要注意，实际上并不是非得使用一个using语句，也可以使用完全限定名。所以，如果省略`using System.Windows.Forms`，仍然可以通过调用`System.Windows.Forms.MessageBox.Show()`来显示一个消息框，编译器就会知道你指的是哪一个命名空间。

**2** **C#程序组织为类。**

每个C#程序都组织为类。一个类可以做任何事情，不过大多数类都只做一件特定的事情。创建这个新程序时，IDE会增加一个名为Form1的类，来显示一个窗体。

```
namespace Contacts
{
  public partial class Form1 : Form
  {
```

将程序命名为Contacts时，IDE将在代码最上面增加一个namespace关键字，为程序创建一个名为Contacts的命名空间。大括号对之间的所有内容都是Contacts命名空间中的一部分。

这是一个名为Form1的类。其中包含绘制窗体以及在窗体上绘制Toolbox控件的所有代码。让IDE创建一个新的Windows Forms Application工程时，IDE就会创建这个类。

**3** **类包含完成具体动作的方法**

类需要做某件事时，会使用一个方法（method）来完成。方法会取一个输入，并完成某个动作，有时还会产生一个输出。一般都采用参数(parameter)形式向方法传递输入。取决于给定的不同输入，方法会有不同的表现。有些方法会生成输出。如果有输出，这个输出称为返回值(return value)。如果看到方法前面有关键字void，则说明这个方法不返回任何结果。

查找匹配的大括号。每个{最后都要与一个}配对。有些括号对会嵌在其他括号对内部。

```
public Form1()
{
   InitializeComponent();
}
```

这行代码调用了一个名为InitializeComponent()的方法，这个方法也由IDE创建。

**4** **语句完成单个动作。**

向程序增加MessageBox.Show()代码行时，就是在增加一条语句（statement）。每个方法都是由语句组成的。程序调用一个方法时，它会执行方法中的第一条语句，然后执行下一条语句，再下一条，如此继续。执行完方法的全部语句或者达到一个return语句时，方法将结束，程序会从原先调用这个方法的语句之后继续执行。

这是一个名为pictureBox1_Click()的方法，用户单击图片框时就会调用这个方法。

这个方法有两个参数，分别名为sender和e。

```
private void pictureBox1_Click(object sender, EventArgs e)
{
  MessageBox.Show("Contact List 1.0", "About");
}
  }
}
```

这是一条语句。你已经知道这条语句会做什么——它会弹出一个小消息框窗口。

这条语句调用了Show()方法，这是MessageBox类的一个方法，这个类在ystem.Windows.Forms命名空间中。

这条语句向Show()方法传入两个参数。第一个参数是将在消息框中显示的一个文本串，第二个参数是在标题栏中显示的串。

# 程序知道从哪里开始

每个C#程序只能有一个入口点方法，而且总命名为Main()。运行程序时它就是由此知道要从哪里开始。

创建这个新的Windows Application解决方案时，IDE增加的文件中有一个名为Program.cs。在Solution Explorer中双击这个文件。其中有一个名为Program的类，这个类有一个`Main()`方法。这个方法正是入口点（entry point），这说明这就是程序中首先要运行的方法。

以下是上一章中IDE自动构建的一些代码。可以在Program.cs中找到这些代码。

代码放大镜

❶
```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Windows.Forms;
```
❷
```
namespace Contacts
{
```
❸
```
    static class Program
    {

        /// <summary>
        /// The main entry point for the application.
        /// </summary>

        [STAThread]

        static void Main()
        {
```
❺
```
            Application.EnableVisualStyles();
```
❹
```
            Application.SetCompatibleTextRenderingDefault(false);

            Application.Run(new Form1());

        }

    }
}
```

所有这些代码的命名空间是Contacts。后面几页将更详细地讨论命名空间。

以斜线开头的代码行是注释，可以把注释增加到你希望的任何地方。这些斜线告诉C#要将这些代码行忽略。

每次运行程序时，都从这里开始，这就是入口点。

这条语句创建并显示Contacts窗体，并在窗体关闭时结束程序。

**严正声明！**

每个类或方法的第一行称为声明。

要记住，这只是深入剖析代码的起点。不过，在进一步深入之前，首先需要了解你看到的是什么。

1. **C#和.NET提供了丰富的内置特性。**

几乎每一个C#类文件的最前面都可以看到类似这样的代码行。System.Windows.Forms是一个命名空间（namespace）。通过using System.Windows.Forms语句，说明该命名空间中的所有类在你的程序中都可用。这种情况下，这个命名空间提供了大量可视化元素，如按钮和窗体。

随着本书内容的展开，学习C#和.NET的其他内置特性时，你的程序将使用越来越多诸如此类的命名空间。

如果没有指定"using"代码行，每次使用这个命名空间中的任何类时都必须显式地加上System.Windows.Forms。

2. **IDE为你的代码选择一个命名空间。**

这是IDE为你创建的命名空间，根据你的工程名，在这里IDE选择Contacts作为命名空间名。程序中的所有代码都在这个命名空间里。

通过使用命名空间，不同程序中允许存在同名（只要这些程序不在同一个命名空间中）。

3. **代码存储在类中。**

这个类名为Program。IDE会创建这个类，并增加启动程序和显示Contacts窗体的代码。

一个命名空间中可以有多个类。

4. **这个代码中只有一个方法，其中包含多条语句。**

命名空间中包含类，类中包含方法。每个方法中则包含一组语句。在这个程序中，这些语句将处理Contacts窗体的启动。动作就是在方法中发生，每个方法都会完成一些动作。

理论上讲，一个程序可以有不只一个Main()方法，而且你可以告诉C#哪一个是入口点……不过现在不需要这么做。

5. **每个程序都有一种特殊的方法，称为入口点。**

每个C#程序必须有且仅有一个名为Main的方法。尽管你的程序中可能有很多方法，但只能有一个最先执行，这就是Main方法。C#会检查代码中的每一个类，查找一个static void Main()方法。之后，运行程序时，这个方法中的第一条语句将得到执行，第一条语句执行之后，其他所有语句依次执行。

**每个C#都必须有且仅有一个名为Main的方法。这个方法就是代码的入口点。**

**运行代码时，Main()方法中的代码最先执行。**

# 可以改变程序的入口点

只要程序有一个入口点，入口点方法在哪个类中或者哪个方法是入口点并不重要。下面删除Program.cs中的Main方法，再创建一个新的入口点。

**动手做！**

把改变方法名时发生的变化记下来，想想看为什么会发生这种情况，把你认为的原因也写下来。

1. 再回到Program.cs，把Main方法重命名为NotMain。现在尝试构建并运行这个程序。会有什么结果？

   ________________________________________

   ________________________________________

在Solution Explorer中鼠标右键单击工程，并选择"Add"和"Class…"。

2. 下面创建一个新的入口点。增加一个名为AnotherClass.cs的新类。要为程序增加一个类，可以在Solution Explorer中鼠标右键单击工程名，并选择"Add"→"Class…"。将这个类文件命名为AnotherClass.cs。IDE将为你的程序增加一个名为AnotherClass的类。以下就是IDE增加的文件：

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

namespace Contacts
{
    class AnotherClass
    {
    }
}
```

为文件增加这4个标准的using代码行。

这个类也放在第一次创建Windows Application工程时由IDE增加的Contacts命名空间中。

IDE自动根据文件名对类命名。

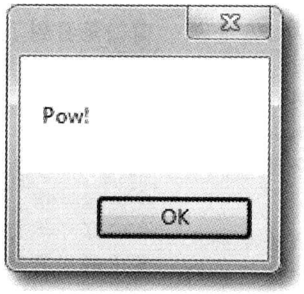
3. 在文件最上面增加一个新的using语句：**using System.Windows.Forms;** 别忘了在这行代码后面加一个分号！

4. 为**AnotherClass**类增加以下方法，把它放在大括号之间：

```
class AnotherClass
{
    public static void Main()
    {
        MessageBox.Show("Pow!");
    }
}
```

MessageBox是System.Windows.Forms命名空间中的一个类，正是因为这个原因，必须增加第3步中的using代码行。Show()是MessageBox类的一个方法。

下面来运行！




## 会发生什么？

不再弹出Contacts应用，现在程序会显示这个消息框。建立这个新的Main()方法时，就为程序提供了一个新的入口点。现在程序所做的第一件事就是运行这个方法中的语句，也就是运行MessageBox.Show()语句。这个方法里没有其他的语句，所以一旦单击OK按钮，程序就会执行完所有语句（只有一条语句），然后结束。

**5** 如何修改程序，让它再弹出Contacts窗口。

提示：只需修改两个文件中的两行代码就能做到。

## Sharpen your pencil

填入注解，分别描述这个C#文件中相应的代码行。我们已经为你填好了其中第一个注解。

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SomeNamespace
{
    class MyClass {
        public static void DoSomething() {
            MessageBox.Show("This is a message");
        }
    }
}
```

C#类需要这些"using"代码行来增加其他命名空间中的方法。

## there are no Dumb Questions

**问：** 大括号有什么用？

**答：** C#使用大括号（{}）将语句组织为代码块（block）。大括号总是成对出现。仅当前面有开始大括号时，才会在后面看到结束大括号。IDE会帮助你完成大括号的匹配，只需单击其中一个（开始大括号或结束大括号），就会看到与之匹配的大括号将加粗显示。

**问：** 我还不是太清楚入口点是什么。可以再解释一次吗？

**答：** 你的程序里有很多很多的语句，不过它们并不会同时运行。程序要从第一个语句开始，先执行这条语句，再执行下一条语句，依次类推。这些语句通常组织为一组类。那么运行程序时，它怎么知道先从哪条语句开始呢？

这就引入了入口点。除非程序中存在且只有一个名为Main()的方法（我们称之为入口点），否则编译器不会构建你的代码。程序就从Main()中的第一条语句开始运行。

**问：** 运行程序时我怎么在Error List窗口里看到了错误？我以为这只会在选择“Build Solution”构建解决方案时才会发生。

**答：** 因为从菜单选择“Start Debugging”或按下工具条上的按钮启动程序运行时，首先会保存解决方案中的所有文件，然后尝试编译。编译代码时（不论是运行之前进行编译，还是构建解决方案时进行编译），如果存在错误，IDE就会在Error List中显示这些错误，而不会继续运行你的程序。

编译代码时显示的大量错误也会出现在Error List窗口中，相应代码下方会出现红色蜿蜒曲线。

## Sharpen your pencil Solution

填入注解，分别描述这个C#文件中相应的代码行。我们已经为你填好了其中第一个注解。

```
using System;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SomeNamespace
{
    class MyClass {
        public static void DoSomething() {
            MessageBox.Show("This is a message");
        }
    }
}
```

C#类需要这些“using”代码行来增加其他命名空间中的方法。

所有代码都放在类中，所以在这里程序需要有一个类。

这个类有一个方法。名为“DoSomething”，调用这个方法时会弹出一个MessgeBox(消息框）。

这是一条语句。执行时，它会弹出一个包含一条消息的小窗口。

# WHAT'S MY PURPOSE?

将IDE生成的以下各个代码段与其完成的功能配对（其中有一些是前面没有介绍过的，可以先猜一猜，看你能不能做对）。

```
partial class Form1
{
  :
  this.BackColor = Color.DarkViolet;
  :
}
```

```
// This loop gets executed three times
```

```
partial class Form1
{
  private void InitializeComponent()
  {
   :
  }
}
```

```
number_of_pit_stopsLabel.Name
        = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
        = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
        = "Number of pit stops:";
```

```
/// <summary>
/// Bring up the picture of Rover when
/// the button is clicked
/// </summary>
```

```
partial class Form1
{
  :
  this.MaximizeBox = false;
  :
}
```

设置一个标签的属性

什么也不做——这只是程序员为读代码的人增加的一个注释，用于对代码做一些解释

禁用Form1窗口标题栏中的最大化图标( )

一种特殊的注释，IDE用这种注释来解释整个代码块完成的工作

改变Form1窗口的背景色

程序打开Form1窗口时就会执行的一个代码块

## WHAT'S MY PURPOSE?

将IDE生成的以下各个代码段与它完成的功能配对（其中有一些是前面没有介绍过的，可以先猜一猜，看你能不能做对）。

```
partial class Form1
{
  :
  this.BackColor = Color.DarkViolet;
  :
}
```

```
// This loop gets executed three times
```

```
partial class Form1
{
  private void InitializeComponent()
  {
  :
  }
}
```

```
number_of_pit_stopsLabel.Name
        = "number_of_pit_stopsLabel";
number_of_pit_stopsLabel.Size
        = new System.Drawing.Size(135, 17);
number_of_pit_stopsLabel.Text
        = "Number of pit stops:";
```

```
/// <summary>
/// Bring up the picture of Rover when
/// the button is clicked
/// </summary>
```

```
partial class Form1
{
  :
  this.MaximizeBox = false;
  :
}
```

设置一个标签的属性

什么也不做—这只是程序员为读代码的人增加的一个注释，用于对代码做一些解释

禁用Form1窗口标题栏中的最大化图标( )

一种特殊的注释，IDE用这种注释来解释整个代码块完成的工作

改变Form1窗口的背景色

程序打开Form1窗口时就会执行的一个代码块

# 两个类可以在同一个命名空间

来看一个名为PetFiler2的程序，它有两个类文件。这里有3个类：Dog类、Cat类、Fish类。由于这些类都在PetFiler2命名空间中，所以Dog.Bark()方法中的语句可以调用Cat.Meow()和Fish.Swim()。各个命名空间和类如何划分到不同文件上并不重要。运行时它们的表现都是一样的。

SomeClasses.cs

```
namespace PetFiler2 {

  class Dog {

    public void Bark() {
      // statements go here
    }

  }

  partial class Cat {

    public void Meow() {
      // more statements
    }

  }
}
```

如果一个类是"public"，则意味着程序中的所有其他类都可以访问这个类的方法。

MoreClasses.cs

```
namespace PetFiler2 {

  class Fish {

    public void Swim() {
      // statements
    }

  }

  partial class Cat {

    public void Purr() {
      // statements
    }

  }
}
```

由于这些类都在同一个命名空间，它们能彼此"看到"（即使位于不同的文件中）。一个类也可以"跨越"多个文件，不过声明这样一个跨多个文件的类时，需要使用"partial"关键字。

只有使用了partial关键字时才能将一个类划分到不同文件中。对于本书中编写的代码，可能不需要这样做，不过要知道，IDE就采用了这种做法将窗体划分到两个文件中，即Form1.cs和Form1.Designer.cs。

关于命名空间和类声明还有很多内容，不过对于现在要完成的工作暂不需要这些知识。可以翻到附录"其他"的第2节了解更多有关内容。

# 程序使用变量处理数据

静下心来想一想，实际上每个程序都是一个数据处理机。有时数据采用文档形式，或者是视频游戏中的一个图像，也可能是一个即时消息。不过，所有这些都是数据。这就引入了变量（variable）。程序要使用变量来存储数据。




**你是不是已经熟悉另外某种语言？**

如果是这样，则你可能会发现这一章中的一些内容看上去很熟悉。不过，还是很有必要花些时间来完成这些练习，因为C#在某些方面与你熟悉的语言可能有所不同。

## 声明变量

声明（declare）一个变量时，要告诉程序这个变量的类型（type）和变量名（name）。一旦C#知道了变量的类型，如果你错误地试图做一些不合理的事情，比如让48353减去“Fido”，你的程序就不能通过编译。

这些是变量类型。

这些是变量名。

```
int maxWeight;
string message;
bool boxChecked;
```

C#使用变量类型来限制这些变量能存储哪些数据。

这些名是为你而指定的。类似于方法和类，要使用有意义而且能描述变量用途的名字。

## 变量会变化

程序运行时，变量在不同时刻会等于不同的值。换句话说，变量的值会变化（正因如此，“变量”这个名字相当贴切）。这非常重要，因为不论是你已经编写的程序还是将要编写的程序，都要以这种思想为核心。所以，如果你的程序将变量`myHeight`设置为等于63：

```
int myHeight = 63;
```

只要代码中出现`myHeight`，C#就都会把它替换为它的值，即63。之后，如果把这个变量值改为12：

```
myHeight = 12;
```

C#将用12来替换`myHeight`，不过变量仍名为`myHeight`。

**程序需要处理数值、文本、*true/false*值或者所有其他类型的数据时，你要用变量维护这些数据。**

## 使用变量前必须先赋值

把以下语句放在一个C#程序中：

```
int z;
MessageBox.Show( "The answer is " + z);
```

试着运行看看。你会得到一个错误，IDE拒绝编译这个代码。这是因为，IDE会检查每个变量，确保你在使用一个变量之前已经为它赋值。要想确保不会忘记对变量赋值，最容易的方法是把声明变量的语句与为变量赋值的语句结合起来：

这些值将赋给变量。

```
int maxWeight = 25000;
string message = "Hi!";
bool boxChecked = true;
```

每个声明都有一个类型，这与前面完全一样。

**如果你编写的代码中使用了尚未赋值的变量，代码就将无法通过编译。通过将变量声明和变量赋值结合到一个语句中，就很容易避免这种错误。**

一旦为变量赋值，这个值就是可以改变的。所以声明变量时为变量赋一个初始值没有任何坏处。

## 一些有用的类型

每个变量都有一个类型，告诉C#这个变量中能存储何种类型的数据。我们将在第4章非常详细地讨论C#中的多种不同类型。同时，我们将重点强调3种最常用的类型。`int`类型保存整数，`string`类型存储文本，`bool`类型存放布尔值true/false。

> **var-i-a-ble, 形容词。**
> 能够改变或调整。
> 利用钻机的变速齿孔，Bob可以根据要完成的任务由慢到快地改变钻机的速度。

# C#使用我们熟悉的数学符号

对于程序员来说，“串”几乎总表示文本串，“int”几乎总是整数的简写。

在变量中存储了一些数据后，又能对它做些什么呢？如果这是一个数字，你可能希望完成加、减、乘或除运算。这样就引入了操作符（operator）。你已经很清楚那些基本的操作符。下面来讨论更多的操作符。以下代码块中使用了一些操作符来完成简单的算术运算：

声明一个新的int变量，名为number，并设置为15。然后将这个变量与10相加。完成第二条语句之后，number将等于25。

第三个语句将改变number的值，将它设置为36乘以15，也就是540。然后，重置这个变量，设置它等于12 - (42 / 7)，这会得到6。

这个操作符稍有些不同，+=表示取number的值，再加上10。由于number目前等于6，加10后将把它的值设置为16。

*= 操作符类似于+=，只不过它将number的当前值乘以3，所以最终number会设置为48。

正常情况下，71除以3等于23.666666…不过由于这里是两个int（整数）相除，因此总会得到一个int结果，23.666…会取整为23。

```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7);
number += 10;
number *= 3;
number = 71 / 3;

int count = 0;
count ++;
count --;
```

我们将大量使用int来完成计数，在这些情况下，++和--操作符会很方便。++使用count自增，也就是将这个值加1，--完成count的自减，将这个值减1，所以最后会等于0。

这个MessageBox会弹出一个消息框，显示“hello again hello”。

对串使用+操作符时，只是将两个串放在一起。它会自动将数字转换为串。

“”是一个空串。其中不包含任何字符（这就像是串类型中的0）。

```
string result = "hello";
result += " again " + result;
MessageBox.Show(result);
result = "the value is: " + count;
result = "";
```

bool存储true或false值。!操作符表示非（NOT）。它将true取反为false，而false取反为true。

```
bool yesNo = false;
bool anotherBool = true;
yesNo = !anotherBool;
```




**现在记不住这些操作符也不必担心。**

你会慢慢了解它们的，因为后面将反复看到这些操作符。

# 使用调试工具查看变量的变化

**动手调试！**

调试工具对于理解程序如何工作非常有帮助。可以用它查看上一页代码的实际运行情况。

**1** **创建一个新的Windows Forms Application工程。**

将一个按钮拖到窗体上，双击这个按钮。输入上一页的所有代码。然后看一看以下截屏图中显示的注释：

创建一个新的Windows Forms Application工程时，这会告诉IDE创建一个包含空窗体和入口点的新工程。你可能希望把它命名为类似"Chapter 2 program 1"。学习这本书的过程中你会构建很多程序。

```
Form1.cs
Chapter_2_Code.Form1                    button1_Click(object sender, EventArgs e)

private void button1_Click(object sender, EventArgs e)
{
    // Here's a great way to use the IDE to see how this code works!
    //
    // First, create a new project in the IDE add a button. Next, double-click on the
    // button so the IDE adds a button1_Click method to your program. Fill in that
    // method with all of the code, starting with "int number = 15;".
    //
    // Now put a breakpoint on the first line by right-clicking on it and choosing
    // "Breakpoint >> Insert Breakpoint". The line should turn red.
    //
    // Next, start debugging your program. You'll see it break on the line where you
    // inserted the breakpoint. Your program's just paused! If you click the Run
    // toolbar button (or hit F5), it will continue. Right-click on "number" and
    // choose "Expression: 'number' >> Add Watch" from the menu. The bottom panel in
    // the IDE should change to the Watches window, and there should be a line in that
    // window for "number". Step through the program line by line using Step Over (F10).
    // You can see the value of the "number" variable change as you go!
    //
    // Do the same for the count, result, yesNo, and anotherBool variables.

    int number = 15;
    number = number + 10;
    number = 36 * 15;
    number = 12 - (42 / 7);
    number += 10;
    number *= 3;
    number = 71 / 3;

    int count = 0;
    count++;
    count--;

    string result = "hello";
    result += " again " + result;
    MessageBox.Show(result);
    result = "the value is: " + count;
    result = "";

    bool yesNo = false;
    bool anotherBool = true;
    yesNo = !anotherBool;
}
100 %
```

注释（以两个或多个斜线开头或者包围在/*和*/记号之间）在IDE中显示为绿色文本。不必操心要在这些记号之间输入什么内容，因为注释总会被编译器忽略。

在一个代码行上设置断点时，这一行会变为红色，而且代码编辑器边缘上的相应位置会出现一个红点。

在IDE中运行代码进行调试时，程序一旦执行到一个断点，就会暂停，允许你检查和修改所有变量的值。

**2** **在第一行代码上插入一个断点。**

鼠标右键单击第一行代码（`int number = 15;`），并从Breakpoint菜单选择"Insert Breakpoint"（也可以单击这行代码并选择"Debug"→"Toggle Breakpoint"或者按下F9）。

**翻开下一页，继续调试！**

3. 开始调试程序。

单击Start Debugging按钮（或者按F5键，也可以从菜单选择“Debug”→“Start Debugging”）在调试工具中运行程序。程序应该正常启动，弹出窗体。

4. 单击按钮，触发断点。

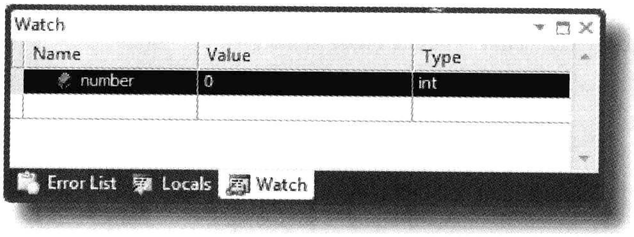
程序一旦到达有断点的代码行，IDE就会自动显示代码编辑器，并用黄色突出显示当前代码行。

```
int number = 15;
number = number + 10;
number = 36 * 15;
number = 12 - (42 / 7)
number += 10;
number *= 3;
number = 71 / 3;
```

调试过程中还可以把鼠标停留在某个变量上，可以看到它的值显示在一个工具提示中……而且可以将其固定一直打开！

5. **为number变量增加一个监视项。**

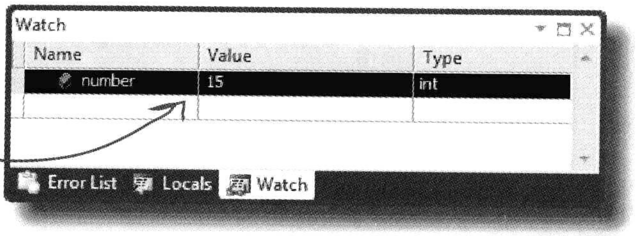
鼠标右键单击number变量（number变量的任何一次出现都可以），从菜单选择“Expression: ‘number’”→“ Add Watch”。Watch窗口会出现在IDE下方的面板中：




**增加一个监视项可以帮助你跟踪程序中的变量值。当你的程序变得越来越复杂时这确实很方便。**

6. **单步跟踪代码。**

按F10键单步跟踪代码（还可以从菜单选择“Debug”→“Step Over”，或者单击Debug工具条上的Step Over按钮）。当前代码行将执行，将number的值设置为15。然后将下一行代码用黄色突出显示，Watch窗口将更新如下：




一旦number变量得到一个新值（15），相应监视项就会更新。

7. **继续运行程序。**

想继续运行程序时，只需按F5键（或者选择“Debug”→“Continue”），程序就会正常地继续运行。

## 循环就是周而复始反复完成一个动作

大多数大型程序都存在这样一种特殊情况：它们几乎都要反复地完成某些特定动作。这就有了循环（loop）的用武之地，循环会告诉你的程序只要某个条件为true（或者为false!），就要不断执行一组特定的语句。

这正是布尔值非常重要的一个主要原因。循环使用条件测试来确定是否继续循环。

```
while (x > 5)
{
    x = x - 3;
}
```

在一个while循环中，只要小括号里的条件为true，就会执行大括号内的所有语句。

每个for循环都有3个语句。第一个建立循环。只要第二个语句为true，循环就会继续。每次循环后将执行第三条语句。

```
for (int i = 0; i < 8; i = i + 2)
{
    MessageBox.Show("I'll pop up 4 times");
}
```

## IDE提示：括号

如果括号不匹配，程序就将无法构建，这会导致一些让人费解的bug。幸运的是，IDE可以帮忙！把光标置于一个括号上，IDE会自动突出显示与它匹配的括号：

```
bool test;
while (test == true)
{
    // Contents of the loop
}
```

## 使用代码片段编写简单的for循环

稍后就会录入一些for循环，而且IDE可以帮助你稍稍加快录入速度。输入for之后再按两次Tab键，IDE会自动为你插入一些代码。如果输入一个新变量，则IDE会自动更新余下的代码片段。再按下Tab键，光标会跳到length。

按Tab键，使光标跳至length。这个循环运行的次数将根据你为length设置的值来确定。可以将length改为一个数字或者一个变量。

```
for (int i = 0; i < length; i++)
{

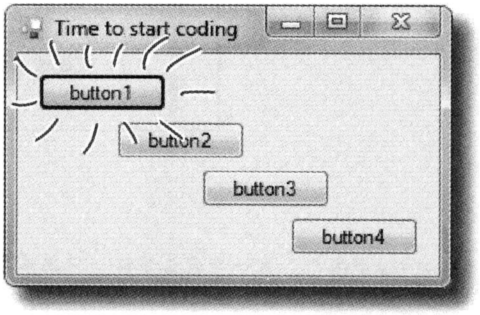
}
```

如果改为其他变量，则代码片段会自动改变该变量的另外两处出现。

## 开始编写代码

所有程序的具体工作都在语句中完成。不过语句并不是凭空独立存在的。所以下面继续深入，来编写一些代码。创建一个新的Windows Forms Application工程。

建立这个窗体




### 一些有帮助的提示

★ 不要忘记所有语句末尾都必须有一个分号：

```
name = "Joe" ;
```

★ 可以在代码前加两个斜线来增加注释：

```
// this text is ignored
```

★ 变量声明包括一个变量名和一个类型（第4章中你将了解大量类型）：

```
int weight;
// weight is an integer
```

★ 类或方法的代码要放在一对大括号内。

```
public void Go() {
    // your code here
}
```

★ 大多数多余的空格都没有影响：

```
int j    =     1234  ;
```

等同于：

```
int j = 1234;
```

## 增加语句来显示一条消息

首先双击第一个按钮。然后向button1 _ Click()方法增加以下语句。仔细察看这个代码，并注意它产生的输出。

x是一个变量。"int"部分告诉C#这是一个整数，这条语句的余下部分将这个变量的值设置为3。

```
private void button1 _ Click(object sender, EventArgs e)
{
    // this is a comment
    string name = "Quentin";
    int x = 3;
    x = x * 17;
    double d = Math.PI / 2;
    MessageBox.Show("name is" + name
        + "\nx is" + x
        + "\nd is" + d);
}
```

有一个名为Math的内置类，它有一个成员，名为PI。Math在System命名空间中，所以这个代码所在文件的最前面需要有一行using System;代码。

\n是一个转义序列，用于向消息框增加一个换行符。



# if/else语句做决策

使用 if/else语句来告诉你的程序，只有当你设置的某个条件（condition）为true（或不为true）时才去做某些事情。很多if/else语句都会检查两个值是否相等。这时就要使用==操作符。它与单个等号(=)操作符有所不同，单个等号用来设置一个值。

```
if (someValue == 24)
{
    MessageBox.Show("The value was 24.");
}
```

每个if语句首先都是一个条件测试。

大括号里的语句只在测试为true的条件下才执行。

要使用两个等号来检查两个值彼此是否相等。

```
if (someValue == 24)
{
    // You can have as many statements
    // as you want inside the brackets
    MessageBox.Show("The value was 24.");
} else {
    MessageBox.Show("The value wasn't 24.");
}
```

if/else语句非常简单。如果条件测试为true，程序将执行第一组大括号之间的语句。否则，就执行第二组大括号之间的语句。




Watch it!

**不要混淆双等号操作符!**

单等号操作符(=)用于设置一个变量的值，双等号(==)则用于比较两个变量。你肯定不相信，程序中有太多的bug都是由于错误地使用了=而不是==造成的（甚至经验丰富的程序员也会犯这种错误）。如果看到IDE向你抱怨“无法将类型‘int’隐式转换为‘bool’”，就可能是这个原因导致的。

# 建立条件，查看条件是否为true

使用if/else语句告诉你的程序，只有当你建立的条件为true（或不为true）时才做某些事情。

**使用一个条件操作符比较两个数字时，这称为条件测试。**

## 使用逻辑操作符检查条件

前面已经见过==操作符，可以用这个操作符来检查两个变量是否相等。另外，还有很多其他操作符。不用担心现在就把所有这些操作符都记住，通过后面几章的学习，你自然会掌握这些操作符：

★ != 操作符的工作与==非常类似，只不过当你比较的两个值不相等时它才为true。

★ 可以使用＞和＜来比较数字，查看其中一个是否大于或小于另一个。

★ ==，!=，＞和＜称为条件操作符。使用这些操作符检查两个变量或值时，称为完成一个条件测试。

★ 可以使用&&操作符或||操作符把单个条件测试结合为一个更长更复杂的测试，&&表示与（AND），||表示或（OR）。所以要检查i是否等于3或者j是否小于5，可以完成以下条件测试：`(i == 3) || (j < 5)`。

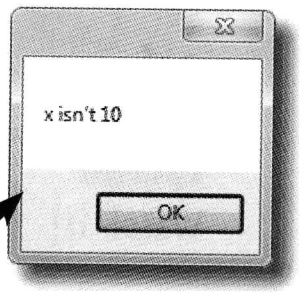
## 设置一个变量，然后检查它的值

做这个工作之前，要确保中止程序的运行。IDE不允许在程序运行的同时编辑代码。可以关闭窗口来停止程序运行，也可以使用工具条上的停止按钮，或者在Debug菜单中选择“Stop Debugging”。

以下是第二个按钮的代码。这是一个if/else语句，检查一个名为x的整数变量，看它是否等于10。

首先建立一个名为x的变量，让它等于5。然后检查它是否等于10。

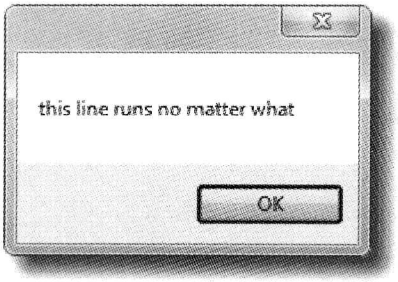
```
private void button2_Click(object sender, EventArgs e)
{
    int x = 5;
    if (x == 10)
    {
        MessageBox.Show("x must be 10");
    }
    else
    {
        MessageBox.Show("x isn't 10");
    }
}
```




**这就是输出。看看能不能修改某行代码，让它显示“x must be 10”。**

## 增加另一个条件测试

第3个按钮会给出以下输出。现在修改两行代码，让它弹出两个消息框。




```
private void button3_Click(object sender, EventArgs e)
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        MessageBox.Show("x is 3 and the name is Joe");
    }
    MessageBox.Show("this line runs no matter what");
}
```

这行代码检查someValue，看它是否小于3，然后再检查name，确保它等于"Joe"。

## 为程序增加循环

以下是最后一个按钮的代码。这里有两个循环。第一个循环是一个**while**循环，只要条件为true就会重复大括号里的语句（当条件为true时做某个事情）。第二个是一个**for**循环。下面来看它是如何工作的。

```
private void button4_Click(object sender, EventArgs e)
{
    int count = 0;

    while (count < 10)
    {
        count = count + 1;
    }

    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }

    MessageBox.Show( "The answer is " + count);
}
```

只要count变量小于10，这个循环就继续重复。

for语句的第二部分是测试。表示"只要i小于5，循环就应继续"。测试在代码块之前运行，只有当测试为true时才会执行代码块。

这会建立循环。它只是为后面循环中要用到的整数赋一个值。

这个语句在每次循环的最后执行。在这里，每次执行循环时，都将使i增1。这称为迭代器，会在代码块中所有语句之后立即运行。

**单击这个按钮之前，再仔细看一遍代码，想想看会显示什么消息框。然后单击按钮，看看你预想的对不对！**

## Sharpen your pencil

下面关于条件测试和循环做一些练习。来看下面的代码。把条件测试圈起来，并填空，使注释能正确地描述所运行的代码。

```
int result = 0; // 这个变量将存放最终结果

int x = 6; // 声明一个变量x，并 设置为6 ........................

while (x > 3) {

  // 只要满足 ................................ 则执行以下语句

  result = result + x; // 将x加至 ................................

  x = x - 1; // ................................ 减去

}

for (int z = 1; z < 3; z = z + 1) {

  // 循环首先 ................................

  // 只要满足 ................................ 则继续循环

  // 每次循环后，................................

  result = result + z; // ................................

}

// 下一条语句将弹出一个消息框，指出

// ................................

MessageBox.Show("The result is " + result);
```

我们已经为你填了第一个空。

## 关于条件测试

可以使用比较操作符检查一个变量的值来完成简单的条件测试。以下是比较两个数字x和y的几种方式：

x < y（小于）
x > y（大于）
x == y（等于，没错，就是用两个等号）

以上是最常用的比较操作符。



等一等!你的逻辑有些问题。如果我写了一个永远也不为false的条件测试，循环会怎么样呢?

**那么，你将永无止境地循环下去!**

每次你的程序运行一个条件测试时，结果都要么为true，要么为false。如果结果为true，程序就会再完成一次循环。每个循环都应当包含一些代码，运行足够的次数后，应当导致条件测试最终返回false。不过，如果没有做到这一点，循环将一直运行下去，直到你强行关闭程序，或者关机!

有时这称为一个无限循环。实际上有时确实希望在程序中使用这样一个循环。

## Sharpen your pencil

以下是几个循环。写出各个循环将无限重复还是会最终结束。如果可以结束，写出它会循环多少次。

**Loop #1**

```
int count = 5;
while (count > 0) {
   count = count * 3;
   count = count * -1;
}
```

**Loop #2**

```
int i = 0;
int count = 2;
while (i == 0) {
   count = count * 3;
   count = count * -1;
}
```

**Loop #3**

```
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
   j = j - i;
   while (j < 25)
   {
       j = j + 5;
   }
}
```

对于Loop #3，这个语句会执行多少次?（指 j = j - i;）

**Loop #4**

```
while (true) { int i = 1;}
```

**Loop #5**

```
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
     p = p * 2;
   }
   q = p - q;
}
```

对于Loop #5，这个语句会执行多少次?（指 q = p - q;）

提示：q开始等于2。想想看迭代器“q = q * 2”何时执行。

要记住，for循环总是在代码块开始时运行条件测试，而在代码块结束时运行迭代器。

## BRAIN POWER

有时可能需要编写一个永远不停止运行的循环，你能想出原因吗（提示：第13章中就会用到一个无限循环)?

## Sharpen your pencil Solution

下面关于条件测试和循环做一些练习。来看下面的代码。把条件测试圈起来，并填空，使注释能正确地描述所运行的代码。

```
int result = 0; // 这个变量将存放最终结果
int x = 6; // 声明一个变量x，并 设置为6
while (x > 3) {
  // 只要满足 x大于3 则执行以下语句
  result = result + x; // 将x加至 result变量
  x = x - 1; // x的值减去1
}
for (int z = 1; z < 3; z = z + 1) {
  // 循环首先 声明一个变量z，并设置为1
  // 只要满足 z小于3 则继续循环
  // 每次循环后， z加1
  result = result + z; // 将z的值增加到result
}
// 下一条语句将弹出一个消息框，指出
// result为18
MessageBox.Show("The result is " + result);
```

这个循环运行两次——第1次z设置为1，第二次z设置为2。达到3时，由于不再小于3，所以循环终止。

## Sharpen your pencil Solution

以下是几个循环。写出各个循环将无限重复还是会最终结束。如果可以结束，则写出它会循环多少次。

**Loop #1**
这个循环执行1次。

**Loop #2**
这个循环会永远运行下去。

**Loop #3**
这个循环执行7次。

**Loop #4**
这也是一个无限循环。

**Loop #5**
这个循环执行8次。

花点时间真正把这个练习做出来。这是一个很好的机会，你可以借此自己试着使用调试工具！在语句q = p - q;上设置一个断点。为变量p和q增加监视项，再单步跟踪循环。

## there are no Dumb Questions

**问：是不是所有代码都放在类中？**

**答：** 不错。只要一个C#程序完成了某个功能，就是因为它执行了一些语句。这些语句要作为类的一部分，而类则属于命名空间。尽管有些东西看上去不是类中的语句（比如使用设计工具设置窗体上一个对象的属性），但是如果查看代码，则你将发现IDE会在类中的某个位置为此增加或修改相关的语句。

**问：有没有一些命名空间名是不能用的？另外哪些命名空间必须使用？**

**答：** 对，确实有一些命名空间名不建议你使用。注意到了吗？C#类文件最上面的所有using代码行都有一个System，这是因为存在一个由.NET Framework使用的System命名空间。在这个命名空间中可以找到所有重要的工具，能够增强程序的功能。如利用System.Data，可以处理数据库表和数据库，另外System.IO允许你处理文件和数据流。不过大多数情况下都可以选择你喜欢的任何名字作为命名空间名（只要其中只包含字母、数字和下划线）。创建一个新程序时，IDE会根据程序名自动选择一个命名空间。

**问：我还是不清楚为什么需要这种部分类。**

**答：** 部分类是指，可以把一个类的代码分开放在多个文件中。IDE创建窗体时就是这么做的，它将你编辑的代码放在一个文件中（如Form1.cs），而它自动修改的代码放在另一个文件中（Form1.Designer.cs）。不过对于命名空间不需要这么做。一个命名空间可以分跨2个、3个甚至数十个文件。只需在文件最前面加上命名空间声明，这个声明之后大括号里的所有内容都将在同一个命名空间中。还要记住一点：一个文件中可以有多个类。另外一个文件中也可以有多个命名空间。后面几章中你将对类有更多了解。

**问：假设我在窗体中拖入一个控件，IDE就会自动生成一组代码。如果我单击“Undo”，则这些代码会怎么样呢？**

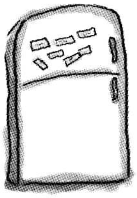
**答：** 要回答这个问题，最好的办法是亲自动手试一试！自己动手看看，做一个动作（IDE将为此生成一些代码），比如在窗体上拖放一个按钮，改变属性等。然后再撤销操作。会发生什么？不错，对于这种简单的情况，你将看到IDE非常聪明，可以自己完成操作的撤销。但是对于更复杂的问题，如向工程增加一个新的SQL数据库，将向你显示一个警告消息。IDE知道怎么撤销这个操作，不过它可能无法重做该操作。

**问：那么，对于IDE自动生成的代码，我是不是要特别仔细？**

**答：** 一般都应当非常仔细。知道IDE如何处理代码会很有用，有时为了解决一个严重的问题还需要知道代码中有些什么。不过，几乎所有情况下都可以通过IDE完成你要做的所有一切。

### BULLET POINTS

- 使用语句告诉程序完成某些动作。语句总是类的一部分，而且每个类都属于一个命名空间。
- 每个语句的最后都要有一个分号(;)。
- 在Visual Studio IDE中使用可视化工具时，IDE会自动在程序中增加或修改代码。
- 代码块用大括号{ }括起。类、while循环、if/else语句和许多其他语句都使用这种代码块。
- 条件测试为true或false。可以使用条件测试来确定一个循环何时结束，还可以用条件测试确定在一个if/else语句中执行哪个代码块。
- 程序需要存储某些数据时，就可以使用变量。=用于完成变量赋值，而==用于测试两个变量是否相等。
- 只要条件测试为true，while循环就会执行其代码块（由大括号定义）内的所有语句。
- 如果条件测试为false，while循环代码块不再运行，执行将移至循环块之后的代码。



# 代码磁贴

冰箱上零乱地贴了一个C#程序的一部分。你能重排这些代码片段，得到一个能正常工作、显示消息框的C#程序吗？一些大括号掉到地上了，因为太小，可能没办法捡起来，所以你可以根据需要自行增加大括号，数目不受限制！

""是一个空串，表示Result中还没有任何字符。

```
string Result = "";
```

这个磁贴没有从冰箱上掉下去……

```
MessageBox.Show(Result);
```

```
if (x == 1) {
Result = Result + "d";
x = x - 1;
}
```

```
if (x == 2) {
Result = Result + "b c";
}
```

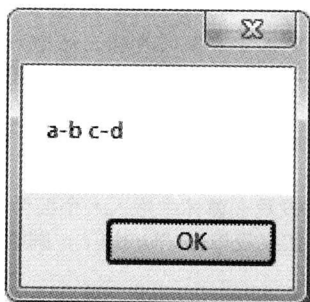
```
if (x > 2) {
Result = Result + "a";
}
```

```
int x = 3;
```

```
x = x - 1;
Result = Result + "-";
```

```
while (x > 0) {
```

输出：




答案见82页。

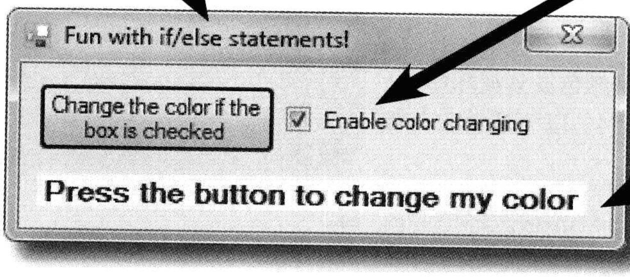
本书中会给出很多像这样的练习。我们会在后面几页给出答案。如果你的思路被卡住了，就不妨先看看答案，这不算偷看！

学习这本书的过程中你会创建很多应用，需要为每个应用指定一个不同的名字。建议根据这一章的章号和窗体标题栏中显示的文本将这个应用命名为“2 Fun with if-else statements”。




来做几个使用if/else语句的练习。你能构建这样一个程序吗？

**这是我们要的窗体。**




**增加这个复选框。**

把它从工具条拖到你的窗体上。使用Text属性改变复选框旁边的文本（还要使用Text属性改变按钮和标签文本）。

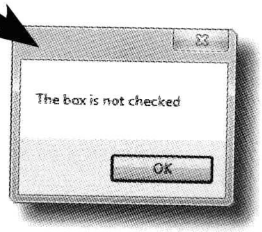
**这是一个标签。**

可以使用属性来改变字体大小，并设置字体为粗体。使用BackColor属性将背景色设置为红色，只需从Web颜色选择中选择“Red”。

**如果用户单击了按钮但是复选框未选中，则弹出这个消息。**

如果复选框名为checkBox1（如果你愿意，也可以改变Name属性），然后利用以下条件测试查看复选框是否选中：

```
checkBox1.Checked == true
```




**如果用户单击了按钮，而且已选中复选框，则改变标签的背景颜色。**

如果标签背景色为红色，则单击按钮时将其改为蓝色。如果它本身是蓝色，则改回为红色。以下语句可以为名为label1的标签设置背景色：

```
label1.BackColor = Color.Red;
```

(提示: 检查标签背景色是否为红色的条件测试与这条语句很相似，不过有一个重要区别！)

## Exercise

下面来点出彩的！先在IDE中创建一个新的Windows Forms Application。

**1** **这是要构建的窗体。**




提示：如果在一个for循环中声明一个变量，例如for (int c = 0; …)，这个变量只在该循环的大括号内部有效。所以，如果有两个for循环都使用了这个变量，那么要么在每个循环中都做出声明，要么循环外应当有一个声明。如果已经在循环外声明了变量c，这两个循环中都不应再做声明。

**2** **让窗体背景呈现迷幻色彩！**

单击按钮时，使窗体的背景色为大量颜色反复循环切换！创建一个循环，其中变量**c**从0变到254。以下是放在大括号中的代码块：

```
this.BackColor = Color.FromArgb(c, 255 - c, c);
Application.DoEvents();
```

这一行告诉程序暂时停止循环，而去做所需的其他工作，比如刷新窗体、检查鼠标单击等。试着去掉这行代码，看看会发生什么。窗体不会自行重绘，因为它在等待直到循环结束，在此之后才会处理这些事件。

对现在来讲，要使用Application.DoEvents()来确保即使在循环中窗体也有反应，不过这种做法有点粗陋，如果不是像这样的玩具型程序，最好不要使用这种代码。在本书后面，你还会学习一种比这好得多的方法，可以让你的程序同一时间做多件事情！

**给我点颜色！**

.NET提供了一组预定义的颜色，如Blue和Red，不过也允许通过指定3个数（一个红色值、一个绿色值和一个蓝色值）使用Color.FromArgb()方法建立你自己的颜色。

**3** **让它慢一点。**

在Application.DoEvents()代码行后面增加这样一行代码，让颜色变化慢下来：

```
System.Threading.Thread.Sleep(3);
```

这条语句在循环中插入一个3毫秒的延迟。这个方法属于.NET库，位于System.Threading命名空间。

**4** **让它平滑一点。**

下面让颜色再循环回到原先的起始颜色。再增加一个循环，其中变量c从254递减到0。大括号里使用同样的代码块。

**5** **持续运行。**

将这两个循环包围在另一个循环中，这个循环将持续执行，不会停止，所以按下按钮后，背景开始改变颜色并一直持续（提示：`while (true)`循环会永远运行下去）。

当一个循环放在另一个循环中时，我们称之为"嵌套"循环。

**唉呀!程序停不下来了!**

在IDE中运行这个程序。开始循环，现在关闭窗口。等一等，IDE并没有回到编辑模式！看起来程序还在运行。需要使用IDE中的停止按钮（方块）才能真正中止程序的运行（或者从Debug菜单选择"Stop Debugging"）。

**6** **让它停下来。**

关闭程序时，要让第5步增加的循环停止。把外循环修改如下：

```
while (Visible)
```

现在运行程序，并单击右角上的X框。窗口会关闭，然后程序也停止了！只不过…… IDE回到编辑模式之前有几秒钟的延迟。

在一个if语句或循环中检查类似Visible的布尔值时，有时会检查（Visible == true），可以去掉"== true"，只包含这个布尔量就足够了。

处理一个窗体或控件时，只要窗体或控件还在显示，Visible就为true。如果将Visible设置为false，将使窗体或控件消失。

提示：&&操作符表示"与"（AND）。采用这种方式，可以把一组条件测试串起来，构成一个大的测试，仅当第一个测试为true，而且第二个测试为true，而且第三个测试也为true，诸如此类，这个测试才为true。这对于解决这种问题非常方便。

**你能看出是什么导致这种延迟吗？能不能修正这个问题，关闭窗口时让程序立即结束？**




来做几个使用if/else语句的练习。你能构建这样一个程序吗？

这是"Fun with If/Else Statements!"练习的完整的Form1.cs文件。如果像这样显示一个文件中的大量代码，我们会在需要由你动手增加的代码下面画一个灰色背景框。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Fun_with_If_Else
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
                if (label1.BackColor == Color.Red)
                {
                    label1.BackColor = Color.Blue;
                }
                else
                {
                    label1.BackColor = Color.Red;
                }
            }
            else
            {
                MessageBox.Show( "The box is not checked" );
            }
        }
    }
}
```

这是窗体的代码。我们将这个解决方案命名为"Fun with If Else"，所以IDE会指定命名空间为Fun_with_If_Else。如果你为解决方案指定了其他名字，就会有一个不同的命名空间。

双击按钮时，IDE为窗体增加一个名为button1_Click()的方法。每次单击按钮时都会运行这个方法。

外部if语句检查复选框，查看它是否选中。选中这个复选框！

内部if语句检查标签的颜色。如标签目前为红色，则执行一条语句把它变成蓝色。

如果标签的背景色不是红色，则运行这条语句，将它设置回红色。

对话框会在复选框未被选中时弹出。

**可以从www.headfirstlabs.com/books/hfcsharp/下载本书所有练习答案的代码。**




下面来点出彩的！

IDE增加这个方法时，它会在大括号前面另外增加一个回车。有时我们会像这样把大括号放在同一行上来节省空间，不过C#并不关心是否有额外空间，所以这是完全合法的。

有时我们不会给出解决方案中的全部代码，而只是给出有改动的部分。FlashyThing工程的所有逻辑都放在这个button1_Click()方法中，在窗体设计工具中双击按钮时IDE会增加这个方法。

要让代码便于阅读，一致性通常非常重要。不过我们有意地采用了多种不同方式，因为你需要习惯于阅读不同人采用不同风格编写的代码。

```
private void button1_Click(object sender, EventArgs e) {
    while (Visible) {
        for (int c = 0; c < 254 && Visible; c++) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
        for (int c = 254; c >= 0 && Visible; c--) {
            this.BackColor = Color.FromArgb(c, 255 - c, c);
            Application.DoEvents();
            System.Threading.Thread.Sleep(3);
        }
    }
}
```

只要窗体可见，外循环会一直运行。一旦窗体关闭，Visible为false，while循环中止。

我们使用了&& Visible而不是&& Visible == true。这就像是在说“如果它可见”而不是“如果它可见是真的”，含义都是一样的。

第一个for循环沿一个方向循环变化颜色，第二个for循环则反向循环变化颜色，使颜色变化看起来很平滑。

通过使用&&操作符使每个for循环还会检查Visible，从而解决存在额外延迟的问题。这样一来，一旦Visible变为false，循环就会结束。

**你能看出是什么导致这种延迟吗？能不能修正这个问题，关闭窗口时让程序立即结束？**

存在延迟是因为while循环检查Visible是否仍为true之前，先要结束for循环。要修正这个问题，可以在每个for循环的条件测试中增加**`&& Visible == true`**。

**你的代码是不是与我们的代码小有出入？条条大路通罗马，解决任何编程问题的方法都不只一种，比如，完全可以使用while循环而不是for循环。如果你的程序能正常工作，就说明你的练习做对了！**

# 池塘谜题



你的任务是从这个池塘里取出一些代码片段，填在代码的空里。不能多次使用同一个代码片段，另外不是所有片段都要用到。你的目标是建立一个能够编译并运行的类。不要以为这很简单，这个问题比看上去要难。

```
int x = 0;
String Poem = "";

while ( __________ ) {

    ______________________________
    if ( x < 1 ) {
        ____________________________
    }
    ______________________________

    if ( __________ ) {

        ____________________________

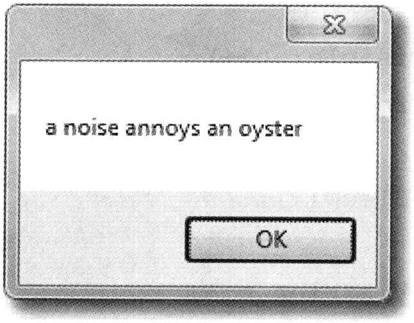
        ___________
    }
    if ( x == 1 ) {

        ____________________________
    }
    if ( __________ ) {

        ____________________________
    }

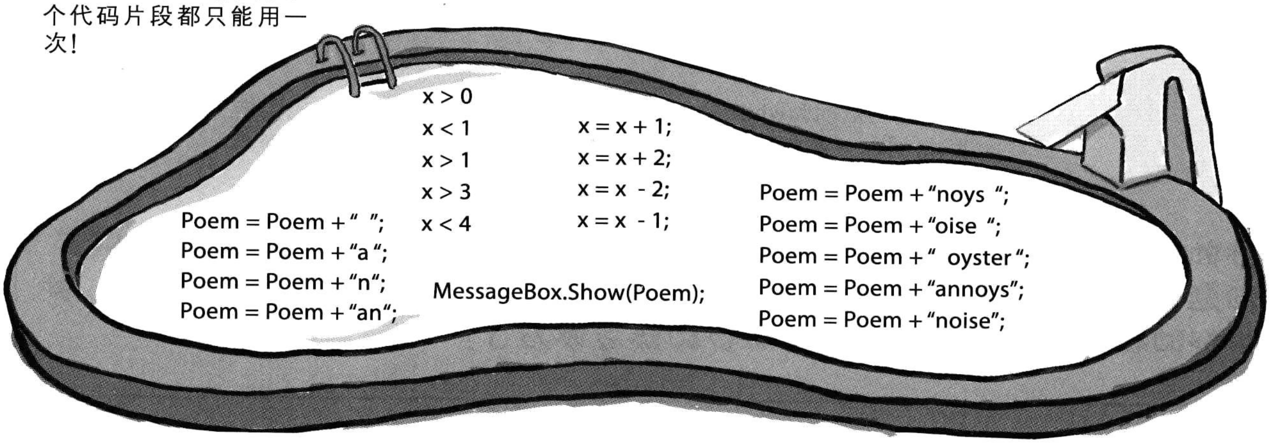
    ____________
}
__________________
```

**输出**




这本书里有很多这种"池塘谜题"练习，就是为了让你充分开动大脑。如果你喜欢有些绕弯子的逻辑问题，肯定会喜欢这种题。如果你不是这种人，不管怎样也试试吧，不过不必担心，允许你看答案，了解该怎么做。如果你被一个池塘谜题难住了，则完全可以跳过去继续看后面的内容。

注意：这个池塘中的每个代码片段都只能用一次！

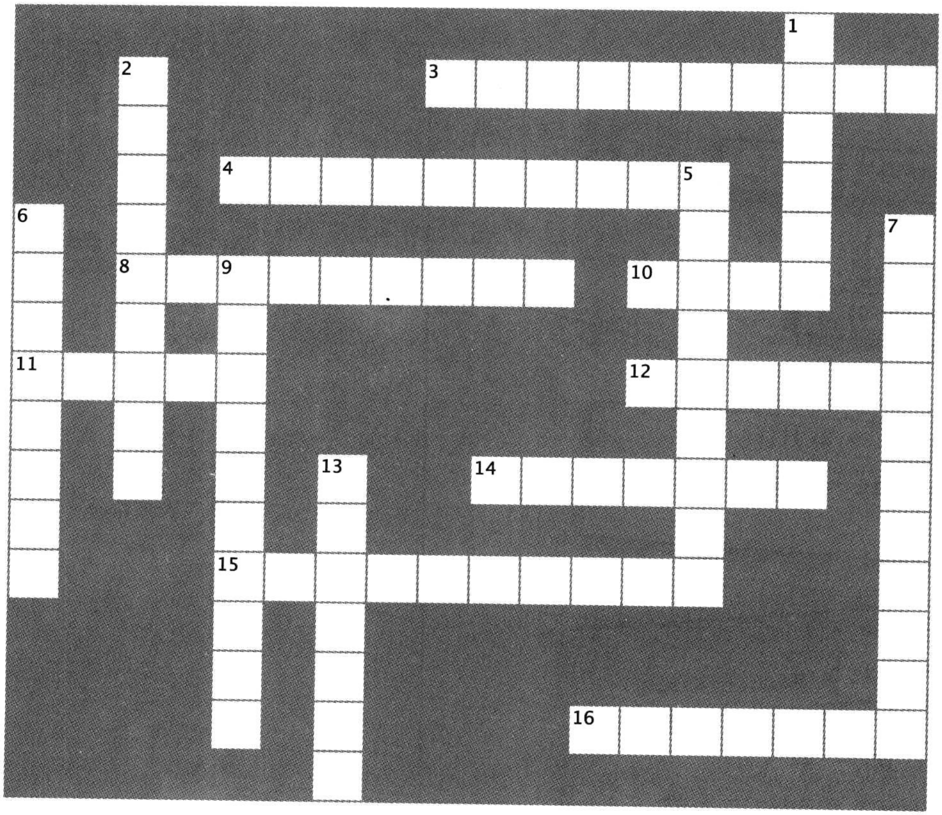





答案见83页。

# CSharp填字游戏

填字游戏怎么来帮你学C#?是这样，这里的所有词都与C#有关，而且都取自这一章。另外线索也是一些脑筋急转弯问题，帮助你以另一种方式将有关的C#概念牢牢地记在你的大脑里。

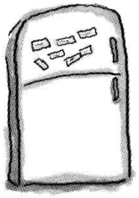



## 横向

3. 使用 __________ 向方法提供信息。
4. button1.Text和checkBox3.Name都是 __________ 的例子。
8. 每条语句最后都以此结束。
10. 每个C#程序的入口点方法名。
11. 包含方法。
12. 语句都放在这里。
14. 这是一种变量，可以为true或false 。
15. 一种特殊的方法，告诉你的程序从哪里开始运行。
16. 这种类分跨多个文件。

## 纵向

1. 方法的输出是其 __________ 值。
2. System.Windows.Forms是 __________ 的一个例子。
5. 完成某个工作的很小一部分程序。
6. 代码块包围在 __________中。
7. 一种测试，告诉循环何时结束。
9. 可以调用 __________.Show()来弹出一个简单的Windows对话框。
13. 包含整数的变量。



# 代码磁贴

冰箱上零乱地贴了一个C#程序的一部分。你能重排这些代码片段，得到一个能正常工作、显示消息框的C#程序吗？一些大括号掉到地上了，因为太小，可能没办法捡起来，所以你可以根据需要自行增加大括号，数目不受限制！

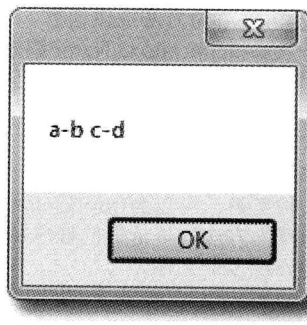
这个磁贴没有从冰箱上掉下去……

```
string Result = "";
int x = 3;
while (x > 0) {
{
    if (x > 2) {
    Result = Result + "a";
    }
    x = x - 1;
    Result = Result + "-";
    if (x == 2) {
    Result = Result + "b c";
    }
    if (x == 1) {
    Result = Result + "d";
    x = x - 1;
    }
}
MessageBox.Show(Result);
```

第一次循环时，x等于3，所以这个条件测试将为true。

第一次循环时，这个语句使x等于2，下一次循环使x等于1。

**输出：**



# 池塘谜题答案



你的任务是从这个池塘里取出一些代码片段，填在代码的空里。你的目标是建立一个能够编译并运行的类。

```
int x = 0;
String Poem = "";

while ( x < 4 ) {

  Poem = Poem + "a";
  if ( x < 1 ) {
    Poem = Poem + " ";
  }
  Poem = Poem + "n";

  if ( x > 1 ) {

    Poem = Poem + " oyster";

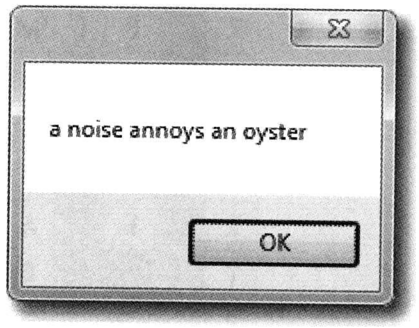
    x = x + 2;
  }
  if ( x == 1 ) {

    Poem = Poem + "noys ";
  }
  if ( x < 1 ) {

    Poem = Poem + "oise ";
  }

  x = x + 1;
}
MessageBox.Show(Poem);
```

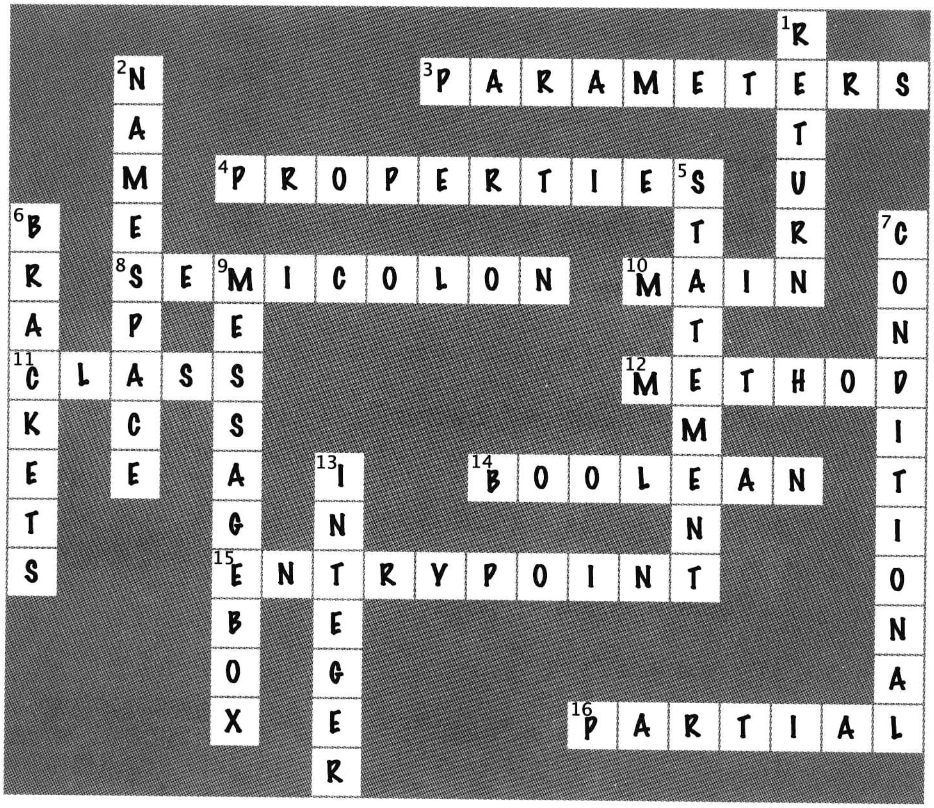
**输出:**




你的答案是不是有所不同？把它键入到IDE中，看看能不能正常工作！池塘谜题往往有不只一个正确答案。

如果真的想挑战，看看能不能得出不同答案！下面给出一个提示：还有一种方案可以让单词片段有序排列。

# CSharp填字游戏答案



# 3 对象：我们的方向！

# 让代码更合理




**你写的每一个程序都要解决一个问题。**

构建一个程序时，首先考虑你的程序要解决什么问题，这通常是个不错的想法。这也说明了为什么对象非常有用。基于对象，你可以根据所要解决的问题建立代码的结构，把宝贵的时间用来考虑需要处理的具体问题，而不是深陷于编写代码的繁杂细节中。如果适当地使用对象，最后不仅写代码轻松，读代码也会很容易，另外还有利于修改代码。

# Mike怎么考虑他的问题

Mike是一个程序员，正打算去参加一个求职面试。他急切地想要展示他高超的C#才能，简直等不及了，不过首先他必须到达面试地点，已经有点晚了！

1. Mike确定前往面试地点的路线。

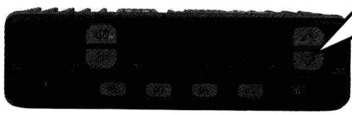


我要上31大道的桥，朝着自由大街向前走，经过布卢姆菲尔德。

Mike先明确目的地，然后想出一个路线。

2. 好在他开着收音机。前面交通严重堵塞，他会因此迟到的！

Mike得到一个新信息，知道需要绕开某条街道。



这里是Frank Loudly向你做“天眼”交通情况报道。看上去自由大街上发生一起三车追尾事故，导致车辆一直堵到第32大道。

3. Mike随即找出一条新路线，以便及时赶到他的面试地点。

现在他找出前往面试地点的一个新路线。

没问题。如果我走28号公路，还是能按时到场的！



# Mike的汽车导航系统怎么考虑他的问题

Mike在车上安装了自己的GPS系统，为他指示行车路线。

这是Mike程序的一个类图。最上面显示了类名，下面是类中的方法。

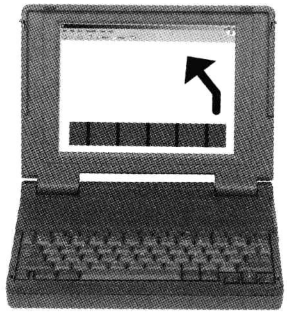
| Navigator |
|---|
| SetCurrentLocation()<br>SetDestination()<br>ModifyRouteToAvoid()<br>ModifyRouteToInclude()<br>GetRoute()<br>GetTimeToDestination()<br>TotalDistance() |

```
SetDestination( "Fifth Ave & Penn Ave" );
string route;
route = GetRoute();
```

导航系统设置一个目的地，并给出一条路线。

以下是GetRoute()方法的输出，这是一个串，其中包含Mike的行车路线。

"Take 31st Street Bridge to Liberty Avenue to Bloomfield"



导航系统得到新信息，知道需要绕开某条街道。

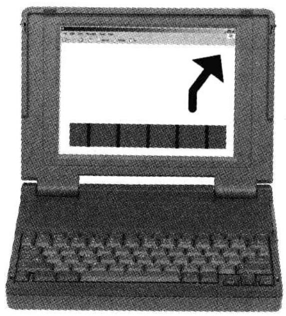
```
ModifyRouteToAvoid( "Liberty Ave" );
```

现在可以给出到达目的地的一条新路线。

```
string route;
route = GetRoute();
```

"Take Route 28 to the Highland Park Bridge to Washington Blvd"

GetRoute()给出一个新路线，不会经过Mike想绕开的那个街道。



**与Mike一样，他的导航系统也采用同样的方式来解决道路导航问题。**

# Mike的Navigator类有一些设置和改变路线的方法

Mike的Navigator类中包含一些方法，具体动作就在这些方法中完成。不过不同于窗体中的button _ Click()方法，这些方法都与一个问题有关：即实现一个城市中的路线导航。也正是因为这个原因，Mike把所有这些方法都放在一个类中，并把这个类命名为Navigator。

Mike精心设计了他的Navigator类，以便轻松地创建和修改路线。要得到一个路线，Mike的程序会调用SetDestination()方法设置目的地，然后使用GetRoute()方法将路线输出到一个串中。如果他需要改变路线，则这个程序会调用ModifyRouteToAvoid()方法调整路线来避开某个街道，然后调用GetRoute()方法得到新路线。

Mike选择了合理的方法名，对于正在考虑如何实现城市路线导航的人来说，这些方法名要便于理解。

```
class Navigator {
    public void SetCurrentLocation(string locationName) { ... }
    public void SetDestination(string destinationName) { ... };
    public void ModifyRouteToAvoid(string streetName) { ... };
    public string GetRoute() { ... };
}
```

这是方法的返回类型。这说明，调用GetRoute()方法的那条语句可以用它来设置一个包含路线的串变量。如果返回类型为void，则说明方法不返回任何结果。

```
string route =
    GetRoute();
```

## 有些方法有返回值

每个方法都由完成某些动作的语句组成。有些方法只是执行这些语句，然后退出。但是另外一些方法还会有一个返回值（return value），也就是在这个方法中计算或生成的一个值，并将这个返回值发送回调用该方法的语句。返回值的类型（如string或int）称为返回类型（return type）。

**return**语句告诉方法要立即退出。如果你的方法没有返回值（这说明声明时方法的返回类型为void），那么return语句就以一个分号结束，并不要求方法中必须有这样一个return语句。不过，如果方法确实有一个返回类型，就必须使用return语句。

这是有返回类型的方法的一个例子，它返回一个int。这个方法使用两个参数（parameter）来计算结果，并使用return语句将这个结果值传回调用该方法的语句。

```
public int MultiplyTwoNumbers(int firstNumber, int secondNumber) {
    int result = firstNumber * secondNumber;
    return result;
}
```

以下是调用一个方法的语句，这个方法将两数相乘，返回一个int:

```
int myResult = MultiplyTwoNumbers(3, 5);
```

方法可以取3和5之类的值。不过也可以使用变量为方法传入值。

## BULLET POINTS

- 类有一些方法，其中包含完成具体动作的语句。可以通过选择合理的方法来设计易于使用的类。
- 有些方法有一个返回类型。要在方法声明中设置方法的返回类型。方法声明如果以“public int”开头，这表示这个方法会返回一个int值。下面是一个返回int值的语句的例子：return 37;
- 方法有返回类型时，必须有一条return语句返回一个与返回类型匹配的值。所以，如果一个方法声明为“public string”，就需要有一条返回一个string的return语句。
- 一旦执行方法中的return语句，程序会立即跳回到调用该方法的语句。
- 并不是所有方法都有一个返回类型。如果方法声明以“public void”开头，就不会返回任何结果。不过仍然可以使用return语句退出一个void方法：if (finishedEarly) { return; }。

## 使用前面所学构建一个使用类的程序

下面将窗体与一个类关联，并让窗体上的按钮调用这个类中的一个方法。

**动手做！**

1. 在IDE中创建一个新的Windows Forms Application工程。然后在Solution Explorer中鼠标右键单击这个工程，并从Add菜单选择“Class...”，在工程中增加一个名为Talker.cs的类文件。将这个新的类文件命名为“Talker.cs”时，IDE会自动将新文件中的类命名为Talker。然后会在IDE的一个新标签页中显示这个新文件。

2. 在类文件最前面增加using System.Windows.Forms;。然后向这个类增加以下代码：

```
class Talker {
    public static int BlahBlahBlah(string thingToSay, int numberOfTimes)
    {
        string finalString = "";
        for (int count = 1; count <= numberOfTimes; count++)
        {
            finalString = finalString + thingToSay + "\n";
        }
        MessageBox.Show(finalString);
        return finalString.Length;
    }
}
```

这个语句声明一个finalString变量，并把它设置为等于一个空串。

这行代码将thingToSay的内容以及末尾的一个换行符（“\n”）增加到finalString变量。

BlahBlahBlah()方法的返回值是一个整数，即所显示消息的总长度。可以对任意一个串增加“.Length”来看它的长度。

这称为属性（property）。每个串都有一个名为Length的属性。计算一个串的长度时，换行符（“\n”）也要算做一个字符。

请继续翻到下一页！

## 你已经构建了什么？

这个新类有一个BlahBlahBlah()方法，它取两个参数。第一个参数是一个串，也就是要说什么。第二个参数是说的次数。调用这个方法时，会弹出一个消息框，其中将消息重复指定的次数。这个方法的返回值是串的长度。这个方法需要一个串作为它的thingToSay参数，另外需要一个数字作为numberOfTimes参数。它将从一个窗体获得这些参数，在这个窗体中，允许用户使用一个**TextBox**控件输入文本，并使用一个**NumericUpDown**控件输入一个数字。

现在来增加一个使用这个新类的窗体！

使用Text属性将TextBox的默认文本设置为“Hello!”。




❸ 让你的工程的窗体如这里所示。

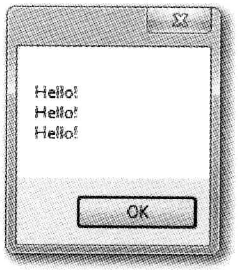
然后双击按钮，让它运行以下代码，调用BlahBlahBlah()并把它的返回值赋给一个名为len的整数：

这是一个NumericUpDown控件。将它的Minimum属性设置为1，Maximum属性设置为10，并将它的Value属性设置为3。

```
private void button1_Click(object sender, EventArgs e)
{
    int len = Talker.BlahBlahBlah(textBox1.Text, (int)numericUpDown1.Value);
    MessageBox.Show("The message length is " + len);
}
```

❹ 现在运行你的程序！单击按钮，可以观察到它会弹出两个消息框。类弹出第一个消息框，然后窗体弹出第二个消息框。

BlahBlahBlah()方法根据其参数值弹出这个消息框。




方法返回一个值时，窗体会弹出这个消息框显示这个值。




**可以向工程增加一个类，与工程中的其他类共享这个类的方法。**

# Mike有一个好主意

面试很成功！不过今天早上的交通堵塞让Mike开始考虑如何改进他的导航系统。



如果能够比较几条路线，找出最快的一条，那该多棒……

## 他可以创建3个Navigator类……

Mike可以复制Navigator类的代码，把它粘贴到另外两个类中。这样他的程序就可以同时存储3个路线了。

这个方框是一个类图（class diagram）。它会列出一个类中的所有方法，这是一种很方便的形式，让人能够一目了然地了解类的全貌。

**Navigator**
- SetDestination()
- ModifyRouteToAvoid()
- ModifyRouteToInclude()
- GetRoute()
- GetTimeToDestination()
- TotalDistance()

**Navigator2**
- SetDestination()
- ModifyRouteToAvoid()
- ModifyRouteToInclude()
- GetRoute()
- GetTimeToDestination()
- TotalDistance()

**Navigator3**
- SetDestination()
- ModifyRouteToAvoid()
- ModifyRouteToInclude()
- GetRoute()
- GetTimeToDestination()
- TotalDistance()

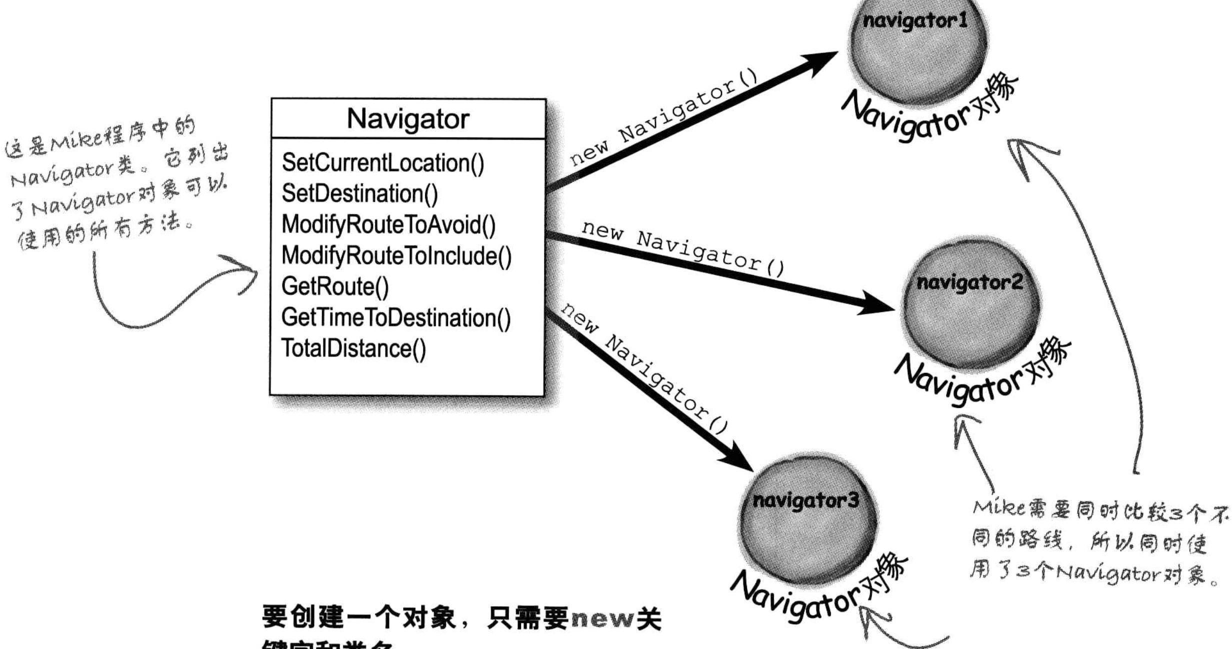


喂，这样不太对劲！如果我想修改一个方法呢？这样一来，我就不得不对3处都分别修改。

**没错！维护同一个代码的3个副本确实非常麻烦。**对于你要解决的大量问题，都需要一种办法能够将一个东西表示多次。在这里，就是有多个路线。不过，当然也可以是多个涡杆、小狗、音乐文件，或者任何东西。所有这些程序都有一个共同点：它们总需要以同样的方式来处理同类事物，而不论这类事物数量有多少。

# Mike可以使用对象来解决他的问题

对象（Object）是C#的一个工具，可以用来处理一组类似的事物。通过使用对象，Mike只需编写一次Navigator类，但在程序中可以根据需要使用多次。



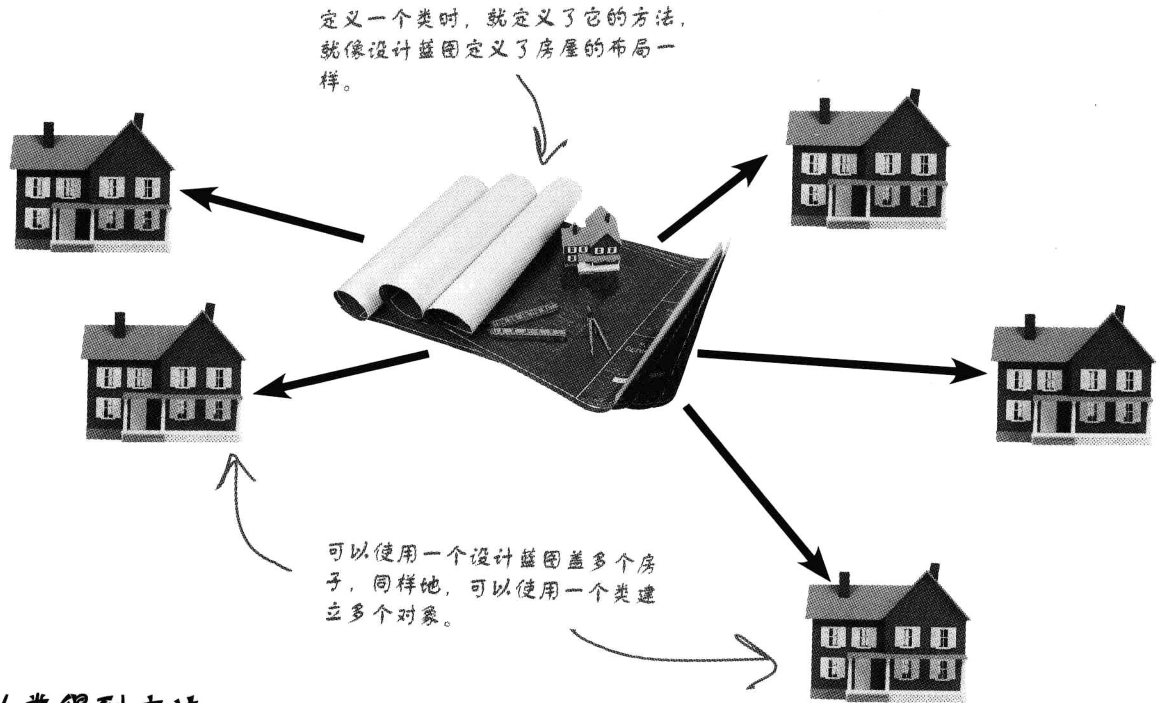

**要创建一个对象，只需要new关键字和类名。**

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination("Fifth Ave & Penn Ave");
string route;
route = navigator1.GetRoute();
```

**现在可以使用对象了！由类创建一个对象时，这个对象会拥有这个类的所有方法。**

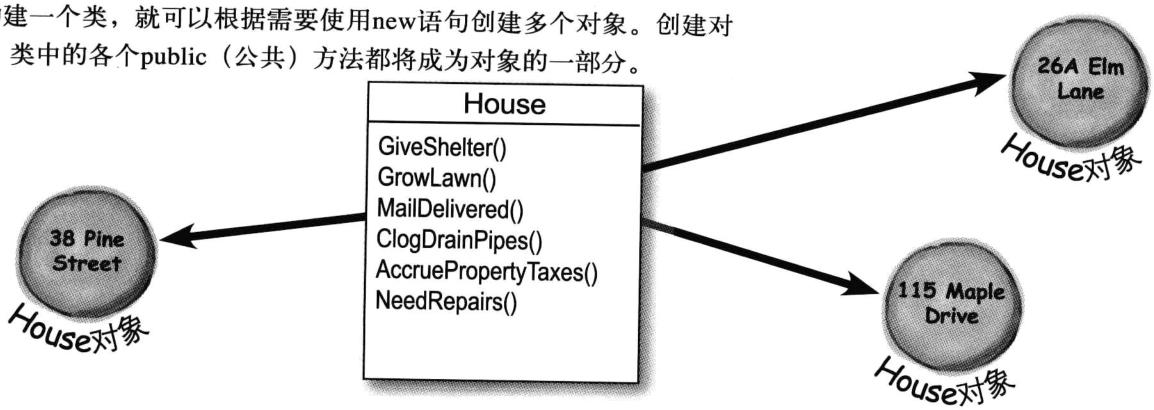
# 使用类来建立对象

类对于对象来说就像是设计蓝图。如果你想在一个郊区房地产开发项目中盖5座相同的房子，肯定不会让设计师画5张同样的设计蓝图。只需要一个蓝图就可以建5个房子。



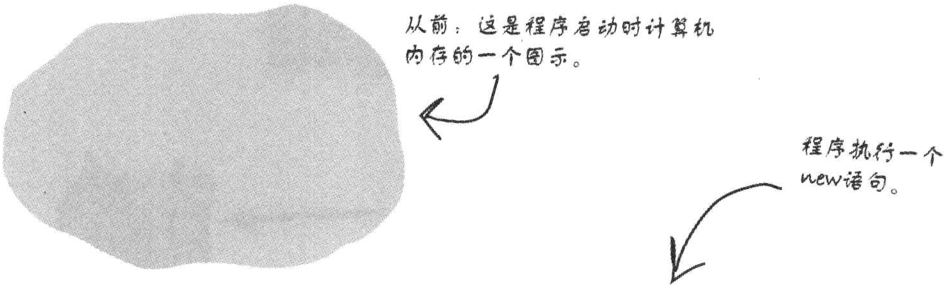

## 对象从类得到方法

一旦构建一个类，就可以根据需要使用new语句创建多个对象。创建对象后，类中的各个public（公共）方法都将成为对象的一部分。

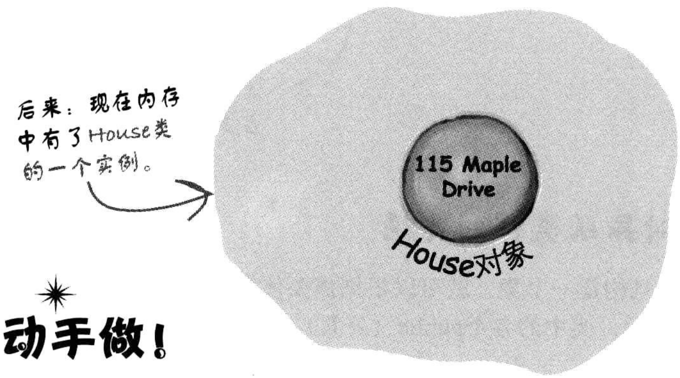


# 由一个类创建新对象时，称为该类的一个实例。

实际上……这一点你已经很清楚了！工具箱里的所有工具都是一个类：Button类、TextBox类、Label类等。将一个按钮从工具箱拖出时，IDE会自动地创建Button类的一个实例，并把它命名为button1。从工具箱拖出另一个按钮时，IDE会创建另一个实例，名为button2。Button的每个实例都有自己的属性和方法。不过，每个按钮的表现都一样，因为它们都是同一个类的实例。




```
House mapleDrive115 = new House();
```




**动手做！**

### 自己动手试试看!

打开任何使用了一个按钮（名为button1）的工程，使用IDE在整个工程中搜索文本“new button1”。你会找到IDE在表单设计工具中为创建Button类实例所增加的代码。

**in-stance, 名词。**

某个事件的一个实例或一次出现。IDE的“查找-替换”特性可以找出一个词的所有实例，并修改为另一个词。

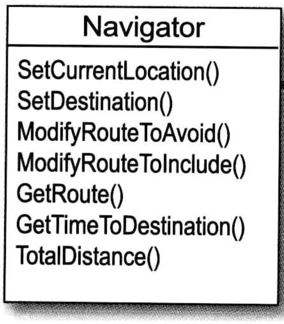
# 更好的解决方案……源于对象！

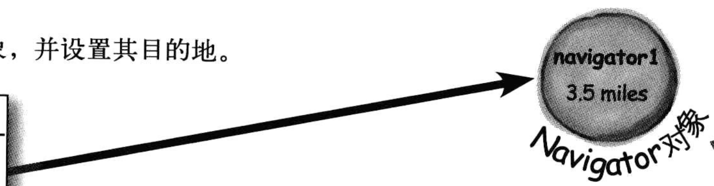
Mike给出了一个新的路线比较程序，通过使用对象，可以在到达相同目的地的3条不同路线中找出最短的一条。构建程序的步骤如下。

GUI代表图形化用户界面（Graphical User Interface），在窗体设计工具中建立窗体时就是在建立GUI。

**1** Mike建立一个GUI，其中有一个文本框textBox1，这个文本框包含3条路线的目的地（destination）。然后增加textBox2，这是某个路线要绕开（avoid）的一条街道，再增加textBox3，这是第3条路线必须包括(include）的一条街道。

**2** 创建一个Navigator对象，并设置其目的地。




navigator1对象是Navigator类的一个实例。

```
string destination = textBox1.Text;
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
route = navigator1.GetRoute();
```

SetDestination()，ModifyRouteToAvoid()和ModifyRouteToInclude()方法都需要一个串作为参数。

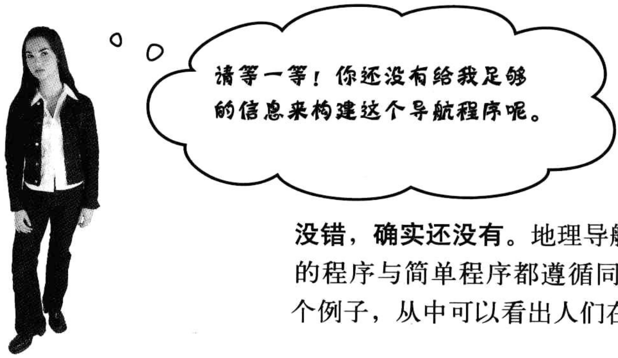
**3** 增加第2个Navigator对象navigator2时，他调用了该对象的SetDestination()方法来设置目的地，然后调用其ModifyRouteToAvoid()方法。

**4** 第3个Navigator对象名为navigator3。Mike设置其目的地，然后调用这个对象的ModifyRouteToInclude()方法。




**从一个类创建新对象时，这称为创建该类的一个实例。**

**5** 现在Mike可以调用各个对象的TotalDistance()方法，得出哪一条路线最短。而且他只需要写一次代码，而不是分别写3次！



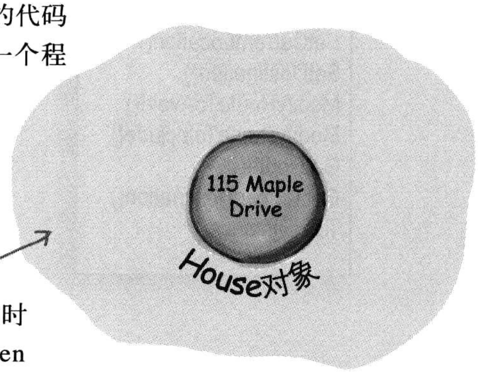

**没错，确实还没有。**地理导航程序构建起来相当复杂。不过复杂的程序与简单程序都遵循同样的模式。Mike的导航程序就是一个例子，从中可以看出人们在现实生活中如何使用对象。

## 理论与实践

谈到模式，在这本书里就你会看到有一种模式将反复出现。我们往往会用几页的篇幅介绍一个概念或思想（如对象），其中用到图片和一些小的代码段来展示这种思想。你要借这个机会后退一步，先不用担心如何让一个程序运行起来，而是要试着了解具体在做什么。

```
House mapleDrive115 = new House();
```

介绍一个新概念（如对象）时，要睁大眼睛，特别注意像这样的图片和代码片段。




介绍一个概念时，我们会给你提供一个机会，把它送入你的大脑。有时我们会利用一个写作练习来学习这个理论，比如下一页上的“Sharpen your pencil”练习。另外一些情况下，我们会直接给出代码。通过理论与实践相结合，这将是一种行之有效的方法，可以让这些概念从书本“走出来”，让它们牢牢记在你的大脑里。

## 关于代码练习的一点建议

如果能记住以下几点，就能更顺利地完成代码练习：

- ★ 很容易遇到语法问题，比如缺少括号或引号。缺少一个大括号可能会带来很多编译错误。
- ★ 有时看看答案比一直被问题所困扰要好得多。受到困扰时，你的大脑就不愿意再学习了。
- ★ 本书中所有代码都经过测试，而且完全可以在Visual Studio 2010中运行！不过很容易犯一些无意的输入错误（比如本该输入一个小写的L，却输入了一个“1”）。
- ★ 如果你的解决方案无法正常编译，可以从Head First Labs网站下载：http://www.headfirstlabs.com/hfcsharp

**完成编码练习时如果遇到问题，不要害怕看答案。还可以从Head First Labs网站下载解决方案。**

## Sharpen your pencil

按上一页Mike的步骤来编写代码，创建Navigator对象并调用这些对象的方法。

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

这里起一个头。这是Mike为了从文本框得到目的地和街道名所写的代码。

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

这些代码用于创建Navigator对象、设置其目的地，并得到距离。

1. 创建navigator2对象，设置其目的地，调用ModifyRouteToAvoid()方法，然后使用它的TotalDistance()方法来设置一个名为distance2的整数变量。

```
Navigator navigator2 = ..............................
navigator2. ..............................
navigator2. ..............................
int distance2 = ..............................
```

2. 创建navigator3对象，设置其目的地，调用ModifyRouteToInclude()方法，然后使用它的TotalDistance()方法设置一个名为distance3的整数变量。

..............................

..............................

..............................

..............................

内置于.NET的Math.Min()方法会完成两个数的比较，并返回其中较小的一个。Mike使用这个方法来找出到达目的地的最短距离。

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```

## Sharpen your pencil Solution

按上一页Mike的步骤来编写代码，创建Navigator对象并调用这些对象的方法。

```
string destination = textBox1.Text;
string route2StreetToAvoid = textBox2.Text;
string route3StreetToInclude = textBox3.Text;
```

这里起一个头。这是Mike为了从文本框得到目的地和街道名所写的代码。

```
Navigator navigator1 = new Navigator();
navigator1.SetDestination(destination);
int distance1 = navigator1.TotalDistance();
```

这些代码用于创建Navigator对象、设置其目的地，并得到距离。

1. 创建navigator2对象，设置其目的地，调用ModifyRouteToAvoid()方法，然后使用它的TotalDistance()方法来设置一个名为distance2的整数变量。

```
Navigator navigator2 = new Navigator()
navigator2.SetDestination(destination);
navigator2.ModifyRouteToAvoid(route2StreetToAvoid);
int distance2 = navigator2.TotalDistance();
```

2. 创建navigator3对象，设置其目的地，调用ModifyRouteToInclude()方法，然后使用它的TotalDistance()方法设置一个名为distance3的整数变量。

```
Navigator navigator3 = new Navigator()
navigator3.SetDestination(destination);
navigator3.ModifyRouteToInclude(route3StreetToInclude);
int distance3 = navigator3.TotalDistance();
```

内置于.NET的Math.Min()方法会完成两个数的比较，并返回其中较小的一个。Mike使用这个方法来找出到达目的地的最短距离。

```
int shortestDistance = Math.Min(distance1, Math.Min(distance2, distance3));
```




**没错！正是因为这个原因，方法中使用了static关键字。**

再来看前面几页构建Talker类时的声明：

```
class Talker
{
  public static int BlahBlahBlah(String thingToSay, int numberOfTimes)
  {
    string finalString = "";
```

调用这个方法时，并没有创建Talker的一个新实例，而只是做了以下调用：

```
Talker.BlahBlahBlah("Hello hello hello", 5);
```

这就是所谓的静态（static）方法，你一直都在用这种静态方法。如果把static关键字从BlahBlahBlah()方法声明中去掉，则必须创建Talker的一个实例才能调用这个方法。除了这个区别外，静态方法与对象方法几乎一样。可以为静态方法传入参数，静态方法也可以返回值，而且静态方法也放在类中。

还可以在另一个地方使用static关键字。可以将整个类都标志为static，这样一来，这个类的所有方法也都必须是静态的（static）。如果想把一个非静态方法增加到静态类中，则将无法编译。

## there are no Dumb Questions

**问：** **考虑“静态”的东西时，我总认为这是一种不会改变的东西。这是不是说，非静态的方法可以改变，而静态方法不能？它们的行为是不是不一样？**

**答：** 不是这样的，静态和非静态方法的行为完全相同。唯一的区别是，静态方法不要求有实例，而非静态方法需要先有一个实例。很多人记不住这一点，因为“静态”这个词确实不太直观。

**问：** **那么，创建一个对象实例之前是不是不能使用类？**

**答：** 可以使用类的静态方法。不过，如果有非静态方法，使用这些方法之前就必须先有一个实例。

**问：** **那么为什么还要这种需要实例的方法呢？为什么不干脆把所有方法都设置为静态方法呢？**

**答：** 这是因为，如果由对象记录某些数据（如Mike的Navigator类实例，每个实例会维护一个不同的路线），就可以使用各个实例的方法来处理这些数据。这样一来，当Mike调用navigator2实例中的ModifyRouteToAvoid()方法时，只会影响存储在这个特定实例中的路线。它不会影响navigator1或navigator3对象。正因如此，他才能同时处理3个不同路线，他的程序才能够跟踪所有这3条路线。

**问：** **那么实例怎么跟踪数据呢？**

**答：** 翻开下一页你就会知道了！

# 实例使用字段来跟踪状态

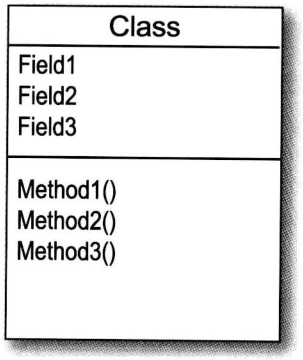
可以在IDE中设置按钮的Text属性来改变按钮上的文本。这样一来，IDE会在设计工具中增加下面的代码：

```
button1.Text = "Text for the button";
```

理论上讲，这是在设置一个属性（property）。属性是一种特殊类型的字段，不过这个问题稍后再来讨论。

现在你知道了button1是Button类的一个实例。这个代码所做的就是修改button1实例的一个字段（field）。可以向类图增加字段，只需在类图中间画一条水平线。字段放在这个水平线的上面，方法放在它下面。

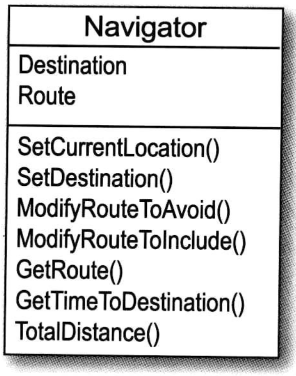
类图中在这里显示字段。类的各个实例使用字段来跟踪其状态。




增加这个水平线将字段与方法分开。

## 方法是对象做什么，字段则是对象知道什么。

Mike创建Navigator类的3个实例时，他的程序就创建了3个对象。每个对象用于跟踪一条不同的路线。程序创建navigator2实例并调用其SetDestination()方法时，要为该实例设置目的地。不过这不会影响navigator1实例或navigator3实例。




Navigator的各个实例知道它的目的地和路线。

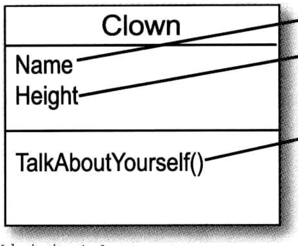
Navigator对象做的事情就是允许设置一个目的地、修改其路线，以及得到该路线的有关信息。

**对象的行为由方法定义，另外使用字段来跟踪其状态。**

# 创建一些实例！

记住，看到一个方法前面有“void”时，说明这个方法不返回任何值。

在类中增加字段很容易。只需在所有方法外部声明变量。现在每个实例都会有其自己的变量副本。




```
class Clown {
  public string Name;
  public int Height;

  public void TalkAboutYourself() {
    MessageBox.Show("My name is "
       + Name + " and I'm "
       + Height + " inches tall.");
  }
}
```

要想创建类的实例，在类声明或方法声明中不要使用static关键字。

记住，*= 操作符告诉C#取左边操作数的值，并乘以右边的操作数。

## Sharpen your pencil

填写执行完相应语句后各消息框中显示的内容。

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

"My name is _______ and I'm ______ inches tall."

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

"My name is _______ and I'm ______ inches tall."

```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

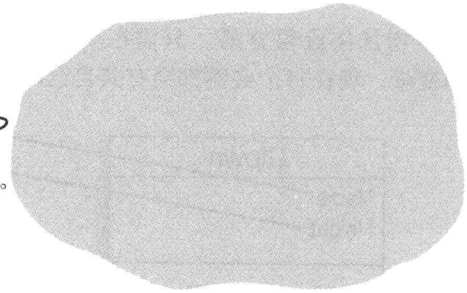
"My name is _______ and I' m ______ inches tall."

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

"My name is _______ and I'm ______ inches tall."

# 感谢内存

程序创建一个对象时，会放在计算机内存中的某一部分，称为堆（heap）。代码用new语句创建一个对象时，C#会立即在堆中预留出空间，来存储该对象的数据。



这是工程启动之前堆的示意图。注意，现在它为空。

下面仔细分析这里发生了什么。

**Sharpen your pencil Solution**

填写执行完相应语句后各消息框中显示的内容。

这些new语句分别创建Clown类的一个实例，在堆中为该对象预留一块内存，并填入该对象的数据。

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

"My name is Boffo and I'm 14 inches tall."

```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

"My name is Biff and I'm 16 inches tall."

```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

"My name is Biff and I'm 11 inches tall."

```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```

"My name is Biff and I'm 32 inches tall."

## 程序创建一个新对象时，将增加到堆中。

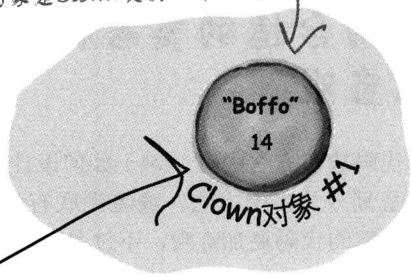
# 程序要做什么

程序中这样创建Clown类的一个新实例：

```
Clown myInstance = new Clown();
```

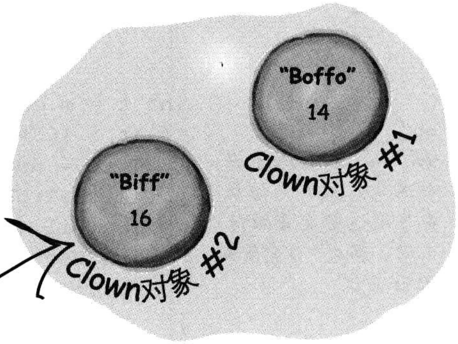
这实际上是将两个语句合并为一个。第一个语句声明一个类型为Clown的变量（Clown myInstance;）。第二个语句创建一个新对象，并把它赋给刚创建的这个变量（myInstance = new Clown();）。执行完以下各语句后，堆的状态如下：

这个对象是Clown类的一个实例。



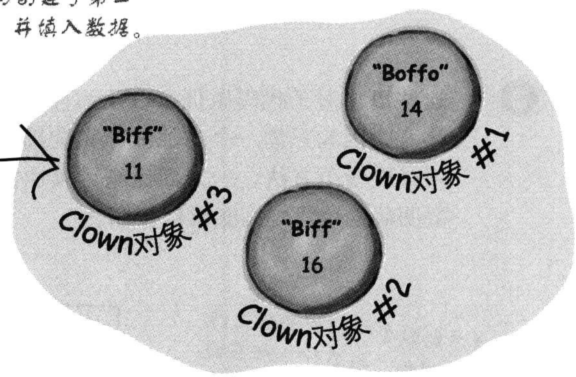

❶
```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```

创建了第一个对象，并设置它的字段。




❷
```
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```

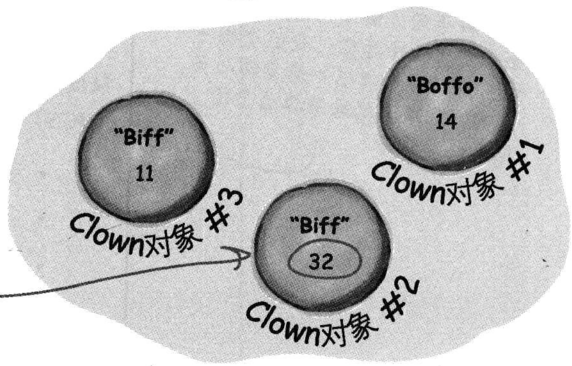
这些语句创建了第二个对象，并填入数据。




❸
```
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```

然后创建第三个Clown对象，并填入数据。

❹
```
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```




没有new命令，这些语句不会创建一个新对象。它们只是修改内存中已有的一个对象。

# 使用合适的类名和方法名使代码更直观

将代码放在方法中时，对于如何设计程序的结构要做出选择。只用一个方法吗？还是划分为多个方法？或者到底有没有必要使用方法？这些选择如果得当，可以使你的代码更加直观，不过另一方面，如果不当心，反而会使代码更费解。

**1** 这是一个相当简洁的代码块，它取自一个制糖机的控制程序。

```
int t = m.chkTemp();
if (t > 160) {
    T tb = new T();
    tb.clsTrpV(2);
    ics.Fill();
    ics.Vent();
    m.airsyschk();
}
```

chkTemp()方法返回一个整数……不过它用来做什么？

"obj"、"ics"和"m"都是很不好的名字！我们无法从中了解这些变量做什么用。另外T()类用来做什么？

clsTrpV()方法有一个参数，不过我们不知道这个参数的含义。

**用点时间仔细看看这个代码。你能看出它的作用吗？**

**2** 这些语句对于代码做什么以及为什么这么做没有给出任何提示。在这个例子中，程序员对结果很满意，因为只需一个方法就能完全搞定。不过让代码尽可能简洁实际上没有多大意义！下面把它划分成多个方法，一方面提高可读性，另一方面保证类中指定的名字都是有意义的。不过，首先要明确这个代码到底要做什么。

怎样才能知道代码要做的工作呢？要明白，所有代码都是因为某种原因才编写的。所以要由你找出这个原因！在这个例子中，可以查看规范手册里的一页说明，程序员就是根据这个规范来编写程序。

**5型通用电子制糖机**
**规范手册**

必须由一个自动系统每3分钟检查一次奶油杏仁糖的温度。如果温度超过160℃，糖块会过热，系统必须完成糖块独立冷却系统（CICS）排水过程。

- 关掉#2涡杆上的阀门。
- 在独立冷却系统中充入大量冷水。
- 排水。
- 检验系统中有没有空气存留。

3 根据手册上的这一页说明，理解前面的代码就容易多了。另外由此还可以得到很多提示，知道如何让代码更易于理解。现在我们知道了为什么要检查变量**t**是否大于160，手册中指出，如果温度超过160° C，意味着奶油杏仁糖会过热。另外，可以看出“**m**”是控制这个制糖机的一个类，其中包含一些静态方法，可以检查糖块温度和空气系统。所以下面把温度检查放在一个方法中，并为类和方法选择更有意义的名字，从而可以直观地看出类和方法的作用。

```
public boolean IsNougatTooHot() {
    int temp = Maker.CheckNougatTemperature();
    if (temp > 160) {
      return true;
    } else {
      return false;
    }
}
```

IsNougatTooHot()方法的返回类型

通过将类命名为“Maker”，将方法命名为“CheckNougatTemperature”理解这个代码会容易得多。

这个方法的返回类型是布尔类型，这说明它会返回一个true或false值。

4 规范中对奶油杏仁糖过热时的处理是怎么规定的？规范指出，要完成糖块独立冷却系统（或CICS）排水过程。所以下面建立另外一个方法，并为“T”类（用于控制涡轮）和“ics”类（控制独立冷却系统，包含两个静态方法分别完成系统的充填和排放）分别选择一个直观的名字：

```
public void DoCICSVentProcedure() {
    Turbine turbineController = new Turbine();
    turbineController.CloseTripValve(2);
    IsolationCoolingSystem.Fill();
    IsolationCoolingSystem.Vent();
    Maker.CheckAirSystem();
}
```

返回类型void说明这个方法不返回任何值。

5 现在代码要直观多了！即使你不知道如果糖块过热需要运行CICS排放过程，从这个代码也能很清楚地看出在做什么：

```
if (IsNougatTooHot() == true) {
    DoCICSVentProcedure();
}
```

**仔细考虑代码所要解决的问题，这可以让你的代码更容易读/写。如果你为方法选择的名字很直观，对于了解这个问题的人来说含义很清楚，理解这个代码……甚至开发代码都会容易得多！**

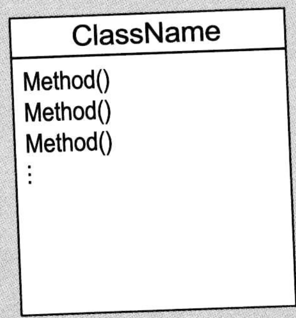
# 为类提供一个自然的结构

花点时间问问自己为什么希望方法是直观的：因为每个程序都要解决一个问题或者都有一个用途。这也许不是一个商业问题，有时一个程序的作用只是为了摆酷或者好玩（如FlashyThing）！但是不论你的程序做什么，代码与所要解决的问题越接近，程序编写起来就越容易（当然读、调整和维护代码……也会更轻松）。

## 使用类图规划类

类图是一种将类画在纸上的简单方法。这个工具意义非凡，利用类图，在编写代码之前你就可以设计代码。

把类名写在类图最上面。然后将各个方法写在下面的框中。现在，一眼就能看到这个类的全部内容。




### 下面来建立一个类图

再来看上一页第5步中的if语句。你已经知道语句总是放在方法中，而方法要放在类中，对不对？在这种情况下，if语句在一个名为`DoMaintenanceTests()`的方法中，这个方法属于`CandyController`类。下面来看这个代码和类图。能看出它们相互之间的关系吗？

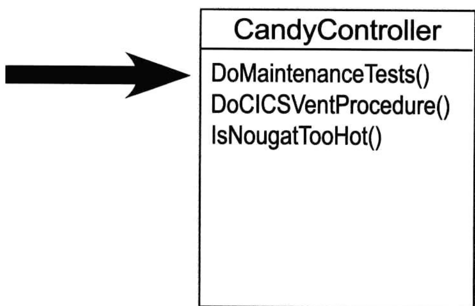
```
class CandyController {

  public void DoMaintenanceTests() {
    ...
    if (IsNougatTooHot() == true) {
       DoCICSVentProcedure();
    }
    ...
  }

  public void DoCICSVentProcedure() ...

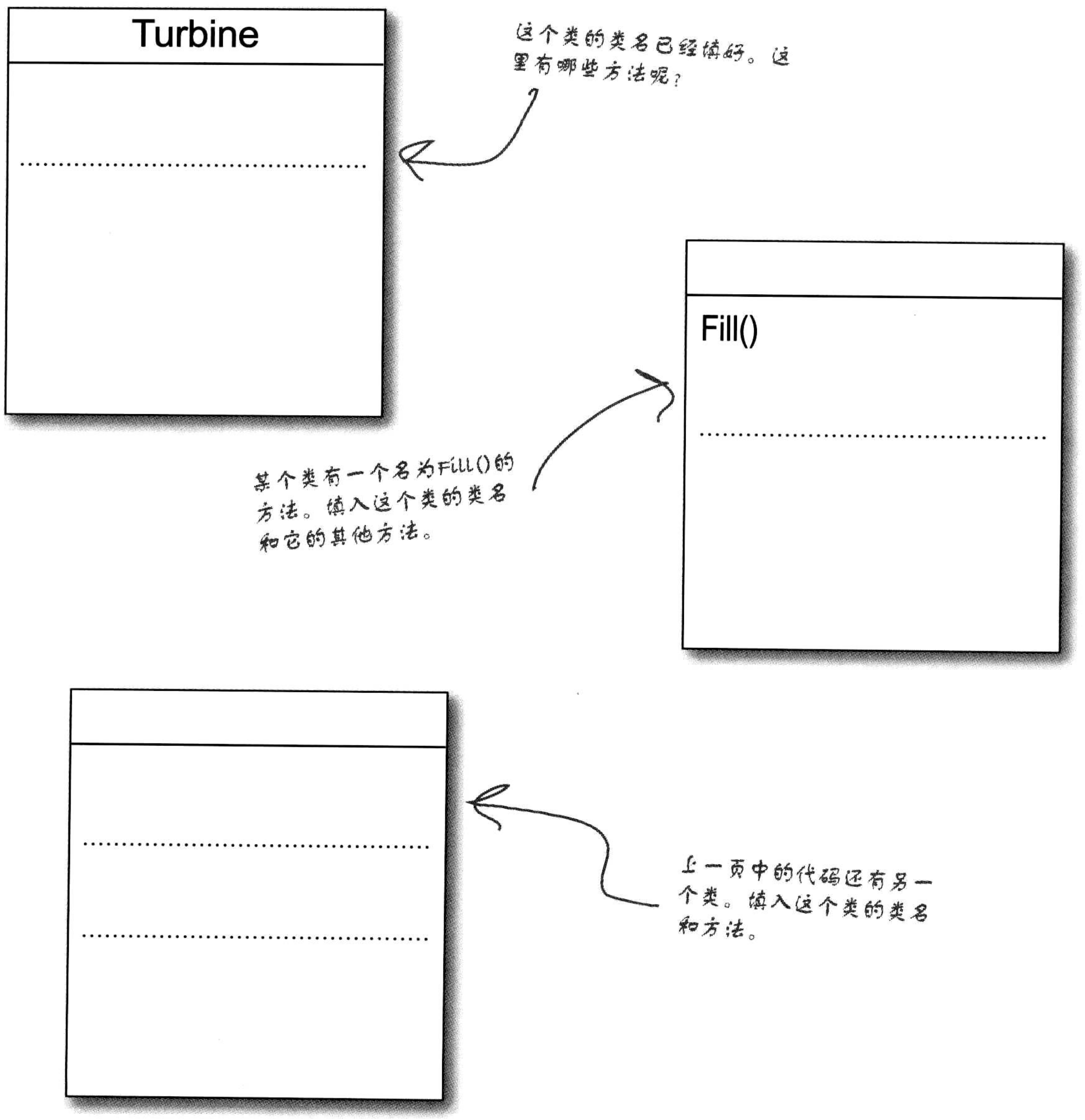
  public boolean IsNougatTooHot() ...

}
```



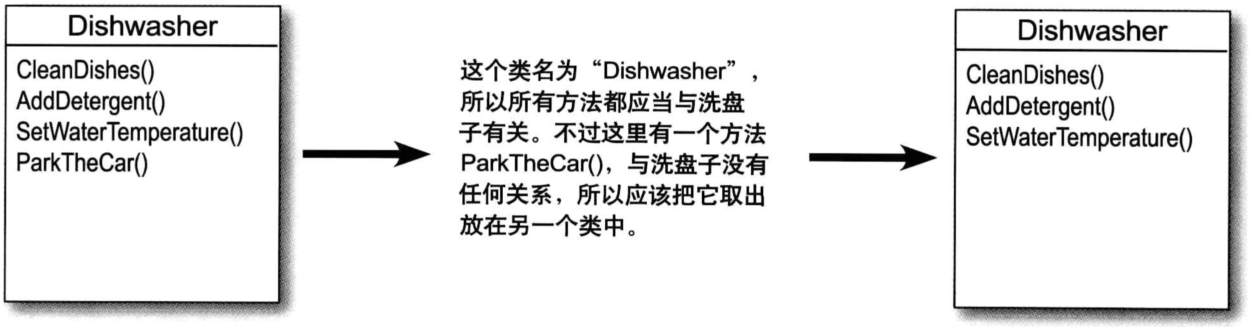
## Sharpen your pencil

上一页构建的制糖控制系统的代码调用了另外3个类。翻回上一页仔细看看代码，填写下面的类图。



# 类图可以帮助你组织类，让它们更有意义

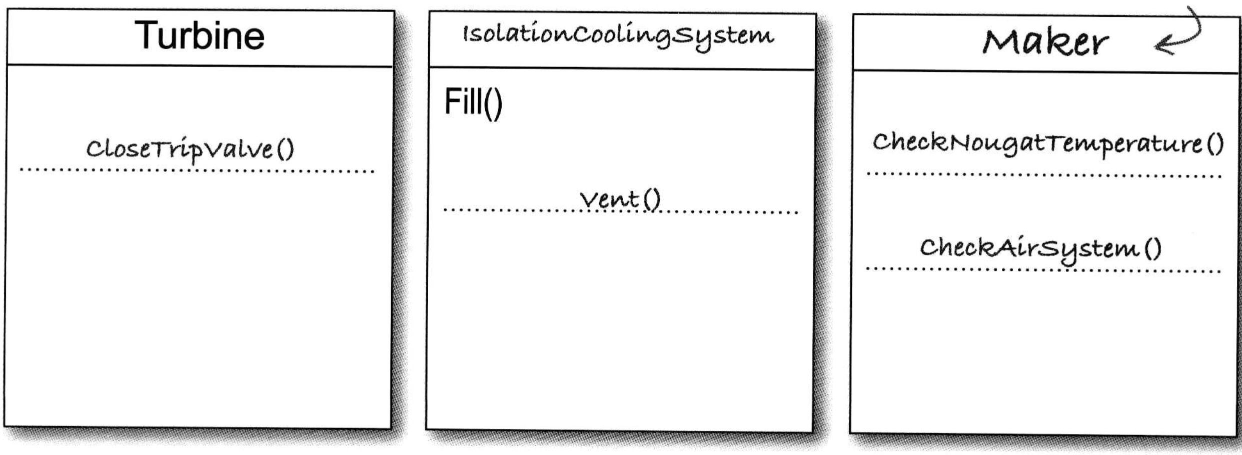
画出类图后，可以在编写代码之前更容易地找出类中可能存在的问题。深入具体细节之前，先从高层考虑你的类，这有助于建立一个更合理的类结构，确保你的代码确实能处理所要解决的问题。类图能让你先向后退一步，确保不会编写一些不必要或者结构很差的类或方法，另外保证你编写的类或方法不仅直观而且易于使用。




## Sharpen your pencil Solution

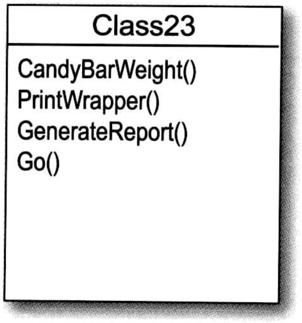
上一页构建的制糖控制系统的代码调用了另外3个类。翻回上一页仔细看看代码，填写下面的类图。

应该可以得出Maker是一个类，因为在Maker.CheckAirSystem()中它出现在点号前面。

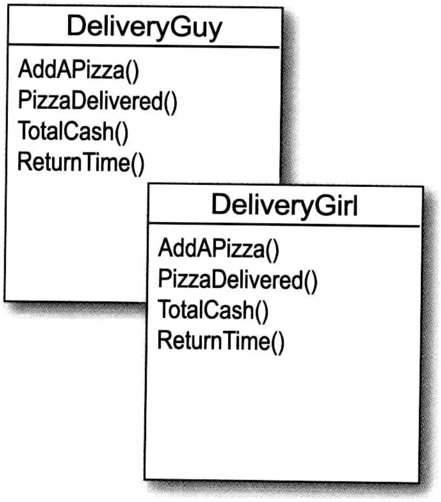


## Sharpen your pencil

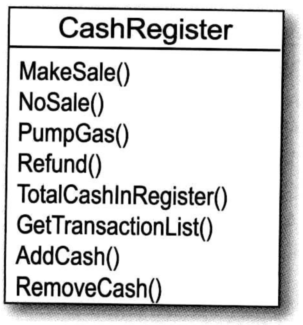
以下每一个类都存在一个严重的设计问题。你认为各个类有什么问题，请写下来，并写出如何修正这个问题。




这个类是前面制糖控制系统的一部分。




这两个类取自一个披萨外卖系统，一家比萨店使用这个系统来跟踪外卖的比萨。




CashRegister类是自动便利店收银系统所用程序的一部分。

## Sharpen your pencil Solution

**我们采用以下方法修正这些类。**解决问题的方法有很多，这里只给出了一种，不过根据类将如何使用，你还可以采用多种其他方法设计这些类。

这个类是前面制糖控制系统的一部分。

类名未能描述出这个类要做什么。程序员看到调用Class23.Go()方法的代码行时，无法知道这行代码能做什么。另外我们还将方法重命名为更具描述性的名字，这里选择了MakeTheCandy()，不过也可以是任何其他名字。

**CandyMaker**

- CandyBarWeight()
- PrintWrapper()
- GenerateReport()
- MakeTheCandy()

这两个类取自一个比萨外卖系统，一家比萨店使用这个系统来跟踪外卖的比萨。

看上去DeliveryGuy类和DeliveryGirl类做的是同样的事情，这两个类都用来跟踪为顾客送外卖比萨的人员。更好的设计是把这两个类用一个类代替，其中增加一个表示性别的字段。

**DeliveryPerson**

- Gender

- AddAPizza()
- PizzaDelivered()
- TotalCash()
- ReturnTime()

这里增加了Gender字段，因为我们认为分别跟踪送外卖的男孩和女孩肯定有某种原因，正是出于这种原因之前才会为他们建立两个类。

CashRegister类是自动便利店收银系统所用程序的一部分。

这个类中的所有方法所做的事情都应该与收银机有关，完成一笔销售、得到交易清单、增加现金……，但只有一个方法除外：打气(PumpGas())。最好把这个方法取出，放到另一个类中。

**CashRegister**

- MakeSale()
- NoSale()
- Refund()
- TotalCashInRegister()
- GetTransactionList()
- AddCash()
- RemoveCash()

```
public partial class Form1 : Form
{
  private void button1_Click(object sender, EventArgs e)
  {
    String result = "";
    Echo e1 = new Echo();
    _________________________
    int x = 0;
    while ( ___________ ) {
      result = result + e1.Hello() + "\n";
      __________________________
      if ( ____________ ) {
        e2.count = e2.count + 1;
      }
      if ( ____________ ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.count);
  }

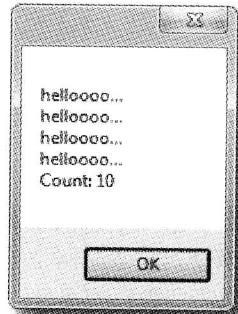
  class ____________ {
    public int _________ = 0;
    public string ___________ {
      return "helloooo...";
    }
  }
}
```

# 池塘谜题



你的任务是从这个池塘里取出一些代码片段，填入代码中的空内。同一个代码片段可以使用多次，另外不是所有片段都要用到。你的目标是建立一个能够编译并运行的类，而且要能生成以下所列的输出。

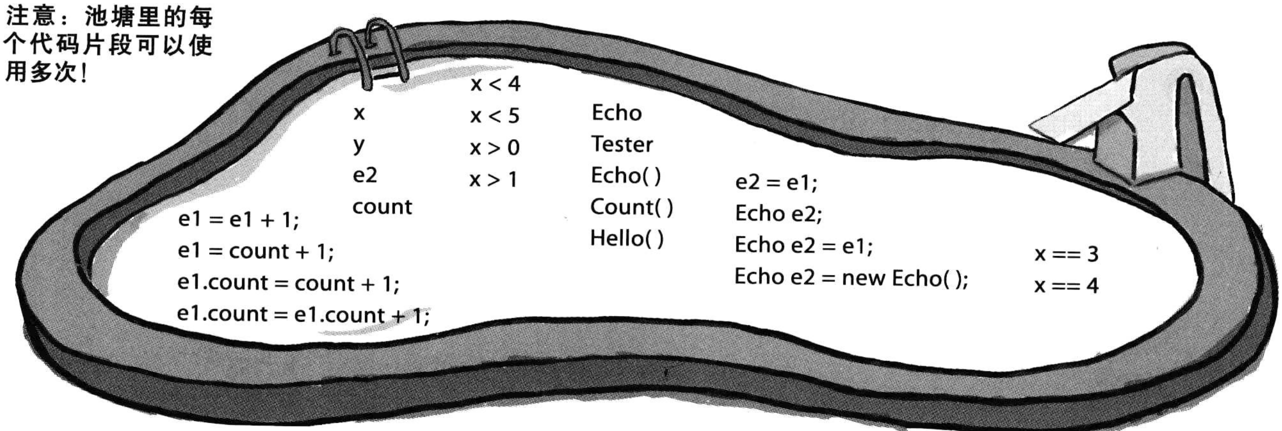
**输出**




**附加问题！**

如果输出的最后一行是24而不是10，怎么完成这个谜题？只需改变一个语句就能做到。

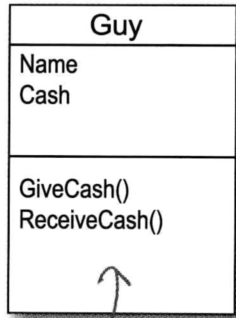
**注意：池塘里的每个代码片段可以使用多次！**




答案见122页。

# 构建一个类来处理人

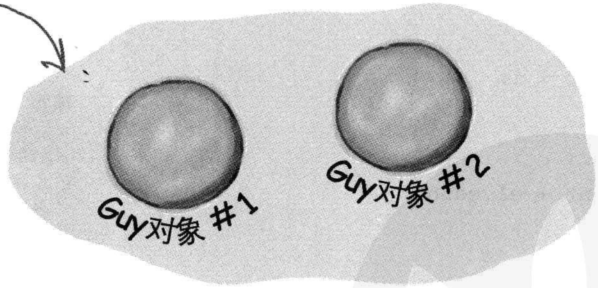
Joe和Bob总是在相互借钱。下面构建一个类来跟踪他们的情况。首先概要说明要构建什么。




我们选择了有意义的方法名。要调用Guy对象的GiveCash()方法告诉他放弃他的部分现金，希望他拿回一些现金时则调用他的ReceiveCash()方法。也可以给这两个方法分别取名为GiveCashToSomeone()和ReceiveCashFromSomeone()，不过这有些太长了！

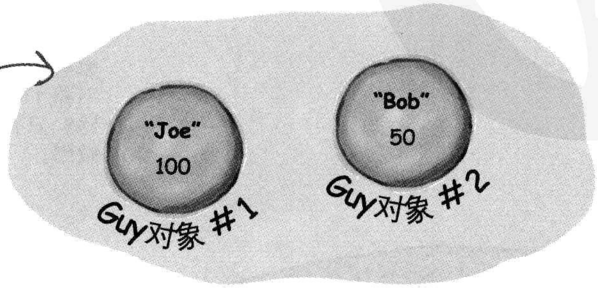
**1** **创建一个Guy类，并向一个窗体增加Guy类的两个实例。**
这个窗体将有两个字段，一个名为joe（跟踪第一个对象），另一个名为bob（跟踪第二个对象）。

创建这两个实例的new语句放在创建窗体时所要运行的代码中。这是加载窗体后雄的状态。




**2** **设置各个Guy对象的cash和name字段。**
这两个对象表示两个不同的人，所以每个人有其自己的名字，口袋里有不同数目的现金。

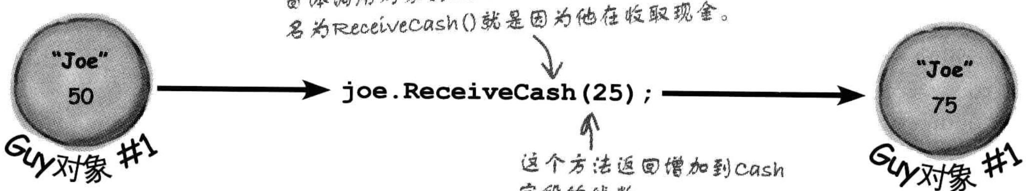
每个人都有一个Name字段来记录名字，另外有一个Cash字段，包含他口袋里的钱数。




对Guy的一个实例调用ReceiveCash()方法时，要传入这个人拿到的现金数额作为参数。所以如果调用joe.ReceiveCash(25)就是告诉Joe收到25美元，并把这些钱放入他的钱包。

**3** **给钱和取钱。**
使用每个人的`ReceiveCash()`方法来增加这个人的现金数额，另外使用`GiveCash()`方法减少他的现金。

窗体调用对象的ReceiveCash()方法。之所以取名为ReceiveCash()就是因为他在收取现金。




这个方法返回增加到Cash字段的钱数。

# 为这些人创建一个工程

**动手做！**

创建一个新的Windows Forms Application工程（因为我们将使用一个窗体）。然后使用Solution Explorer增加一个新类，名为Guy。一定要在Guy文件最前面增加“using System.Windows.Forms;”。然后填入Guy类的内容。代码如下：

```
class Guy {
    public string Name;
    public int Cash;

    public int GiveCash(int amount) {
        if (amount <= Cash && amount > 0) {
            Cash -= amount;
            return amount;
        } else {
            MessageBox.Show(
             "I don't have enough cash to give you " + amount,
             Name + " says...");
            return 0;
        }
    }

    public int ReceiveCash(int amount) {
        if (amount > 0) {
            Cash += amount;
            return amount;
        } else {
            MessageBox.Show(amount + " isn't an amount I'll take",
                  Name + " says...");
            return 0;
        }
    }
}
```

Guy类有两个字段。Name字段是一个串，其中包含这个人的名字（“Joe”）。Cash字段是一个int，其中跟踪这个人口袋里有多少钱。

GiveCash()方法有一个参数名为amount，要用这个参数告诉这个人需要给你多少钱。

使用一个if语句检查是否有足够的钱，如果有，则从口袋里拿出，并作为返回值返回。

Guy要保证你要的钱数是正数，否则这个人的钱不会减少反而会增加。

如果这个人没有足够的现金，会用一个消息框告诉你，并让GiveCash()返回0。

ReceiveCash()方法的工作与GiveCash()方法很类似。也要传入一个amount作为参数，同样要检查amount来确保指定的钱数大于0，然后将这个人的现金增加指定的数额。

如果amount是正数，ReceiveCash()会返回增加的数额。如果amount为0或是一个负数，则会显示一个消息框，然后返回0。

}　要当心你的大括号。大括号的数目很容易弄错，要确保每一个开始大括号都有一个与之匹配的结束大括号。如果二者一致，当你输入最后的结束大括号时，IDE会自动将它们缩进。

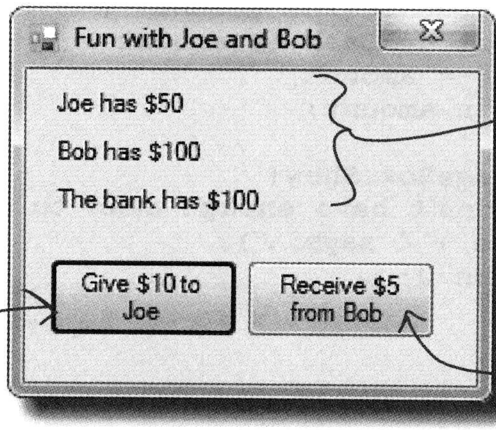
# 建立一个窗体与这些人交互

Guy类很棒，不过这还只是一个开始。现在建立一个窗体，其中使用Guy类的两个实例。窗体中有一些标签指示两个人的名字，以及他们有多少钱，另外还有一些按钮来完成给钱和拿钱的动作。

动手构建！

**1** **为窗体增加两个按钮和3个标签。**

窗体上面的两个标签显示了每个人有多少现金。我们还为窗体增加了一个名为bank的字段变量，第3个标签就显示了银行里有多少钱。你要对拖到窗体上的标签命名。可以单击你想命名的各个标签，并在Properties窗口中修改“(Name)”行的值。这样能使你的代码更可读，因为这样一来你可以使用“joesCashlabel”和“bobsCashlabel”而不是“label4”和“label5”。




将最上面的标签命名为joesCashLabel，它下面的标签命名为bobsCashLabel，另外将最底下的标签命名为bankCashLabel。不用改变它们的Text属性，我们将为窗体增加一个方法来设置这些标签的文本。

这个按钮将调用Joe对象的ReceiveCash()方法，传入10作为amount参数值，并从窗体的bank字段减去Joe收到的钱数。

这个按钮将调用Bob对象的GiveCash()方法，传入5作为amount参数值，并把Bob给的钱数加至窗体的bank字段。

**2** **为窗体增加字段。**

这个窗体需要跟踪两个人的情况，所以对于每个人需要有一个字段。分别命名为joe和bob。然后向窗体增加一个名为bank的字段，跟踪给出和收到的钱数。

```
namespace Your_Project_Name {
    public partial class Form1 : Form {
        Guy joe;
        Guy bob;
        int bank = 100;

        public Form1() {
            InitializeComponent();
        }
```

因为我们使用Guy对象来跟踪Joe和Bob，所以要使用Guy类声明窗体中的相应字段。

根据窗体交给Guy对象以及从Guy对象得到的数额，bank字段的钱数会上下变化。

**3** **为窗体增加一个方法来更新标签。**

窗体上面的标签显示了每个人有多少钱，以及bank字段中有多少钱。所以要增加一个UpdateForm()方法设置标签，使之反映最新状态，要确保返回类型为void，这是为了告诉C#该方法不返回任何值。在前面为窗体增加bank字段的代码下面键入这个方法：

```
public void UpdateForm() {
    joesCashLabel.Text = joe.Name + " has $" + joe.Cash;
    bobsCashLabel.Text = bob.Name + " has $" + bob.Cash;
    bankCashLabel.Text = "The bank has $" + bank;
}
```

注意如何使用Guy对象的Name和Cash字段更新标签。

这个新方法很简单。它只是通过设置Text属性来更新3个标签。可以让按钮调用这个方法，保持标签反映最新状态。

**4** **双击各个按钮，增加代码与这些对象交互。**

让左边的按钮名为button1，右边的按钮名为button2。然后双击这两个按钮，双击各按钮时，IDE会向窗体增加两个方法，名为button1 _ Click()和button2 _ Click()。为这两个方法增加以下代码：

```
private void button1 _ Click(object sender, EventArgs e) {
    if (bank >= 10) {
        bank -= joe.ReceiveCash(10);
        UpdateForm();
    } else {
        MessageBox.Show("The bank is out of money.");
    }
}
```

用户点击"Give $10 to Joe"按钮时，窗体调用Joe对象的ReceiveCash()方法，但只有在银行有足够的钱时才能调用。

银行至少需要有$10交给Joe。如果没有足够的钱，会弹出这个消息框。

```
private void button2_Click(object sender, EventArgs e) {
    bank += bob.GiveCash(5);
    UpdateForm();
}
```

"Receive $5 from Bob"按钮不需要检查银行有多少钱，因为它只会加上Bob交回的钱。

如果Bob没钱了，GiveCash()将返回0。

**5** **开始时Joe有$50，Bob有$100。**

由你来确定开始时如何为Joe和Bob适当地设置Cash和Name字段。把有关代码放在窗体的InitializeComponent()下面，这是一个由设计工具生成的方法，只是在窗体第一次初始化时运行一次。完成后，将两个按钮单击多次，确保单击一个按钮时会从银行取出$10交给Joe，单击另一个按钮时会从Bob拿走$5，增加到bank。




```
public Form1() {
    InitializeComponent();
    // Initialize joe and bob here!
}
```

在这里增加代码，创建这两个对象，并设置其Name和Cash字段。




由你来确定开始时如何为Joe和Bob适当地设置Cash和Name字段。把有关代码放在InitializeComponent()的下面。

```
public Form1() {
    InitializeComponent();

    bob = new Guy();
    bob.Name = "Bob";
    bob.Cash = 100;

    joe = new Guy();
    joe.Name = "Joe";
    joe.Cash = 50;

    UpdateForm();
}
```

在这里建立Guy的第一个实例。第一行创建对象，后面两行设置其字段。

然后对Guy类的第二个实例做同样的处理。

确保调用UpdateForm()，使窗体第一次弹出时标签显示正确。

现在要保存这个工程，后面几页还会用到这个工程。

## there are no Dumb Questions

**问：为什么解决方案一开始不是“Guy bob = new Guy()”？为什么去掉了第一个“Guy”？**

**答：** 因为你已经在窗体最前面声明了bob字段。应该记得吧？语句“int i = 5;”等价与“int i”和“i = 5;”这两条语句。这里也是一样。也可以像这样在一行上声明bob字段：“Guy bob = new Guy();”。不过由于窗体最前面已经有了这个语句的前半部分(“Guy bob;”)，所以这里只需它的后半部分，也就是设置bob字段，创建 Guy()的一个新实例。

**问：既然如此，那为什么不去掉窗体最前面的那行“Guy bob;”呢？**

**答：** 这样一来，名为bob的变量就只能存在于这个特殊的“public Form1()”方法中了。在方法中声明一个变量时，它将只在这个方法中有效，不能从其他方法访问这个变量。不过，如果在方法之外声明（不过仍在窗体或类中），就可以从该窗体或类中的任何其他方法访问这个变量。

**问：如果去掉第一个“Guy”会怎么样？**

**答：** 会遇到麻烦，窗体无法正常工作，因为没有设置窗体的bob字段。仔细想想，你会发现为什么会这样。如果在窗体最前面有以下代码：

```
public partial class Form1 : Form {
    Guy bob;
```

然后在某个方法中有下面的代码：

```
Guy bob = new Guy();
```

这就声明了两个变量。这有些让人糊涂，因为它们是同名的。不过其中一个在整个窗体中都有效，而另外一个(就是你增加的新变量）仅在这个方法中有效。下一行代码(bob.Name = “Bob”;）只会更新这个局部（local）变量，而不会影响窗体中的那个变量。所以运行这个代码时，会给出一条讨厌的错误消息(“NullReferenceException not handled”)，说明你在使用new创建一个对象之前就试图使用这个对象。

## 还有更容易的方法来初始化对象

我们创建的几乎每一个对象都需要以某种方式初始化。Guy对象也不例外，在设置它的Name和Cash字段之前，这个对象没有任何用处。由于初始化对象字段的工作如此常见，所以C#为此提供了一种简便方法，称为对象初始化方法（object initializer）。IDE的智能提示（IntelliSense）可以帮助你编写这种对象初始化方法。

**对象初始化方法可以节省你的时间，使代码更紧凑，更可读……而且IDE会帮助你编写对象初始化方法。**

1 这是原来为初始化Joe的Guy对象所写的代码。

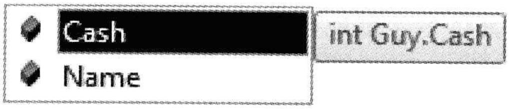
```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

2 删除后面两行代码，并删除“Guy()”后面的分号，增加一个右大括号。

```
joe = new Guy() {
```

3 按空格。一旦按下空格，IDE会弹出一个智能提示窗口，显示出能够初始化的所有字段。

```
joe = new Guy() {
```

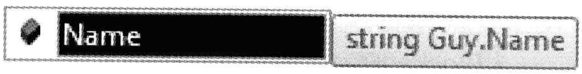



4 按Tab键，告诉IDE增加Cash字段，然后设置这个字段等于50。

```
joe = new Guy() { Cash = 50
```

5 输入一个逗号，一旦输入逗号，就会显示另一个字段。

```
joe = new Guy() { Cash = 50,
```




5 完成这个对象初始化方法。现在你可以少写两行代码！

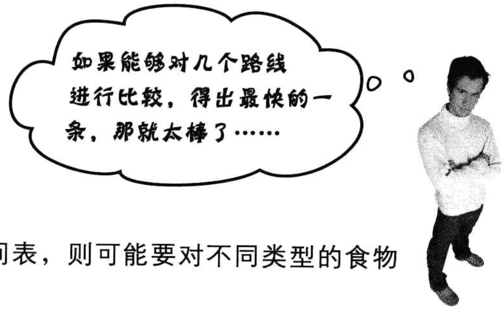
```
joe = new Guy() { Cash = 50, Name = "Joe" };
```

这个新声明与原来写的3行代码完成的工作是一样的。只是更为简短，更可读。

# 关于设计直观类的一些想法

## ★ 构建程序是为了解决一个问题

花点时间来考虑这个问题。它能很容易地划分成多个部分吗？如何向其他人解释这个问题？这些都是设计类时需要考虑的一些很好的问题。



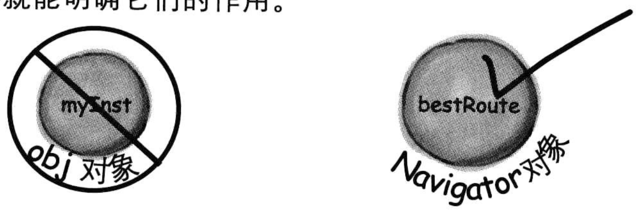

## ★ 你的程序用到了哪些实际事物？

如果一个程序用来帮助动物园管理员记录动物的喂养时间表，则可能要对不同类型的食物和不同类型的动物建立类。



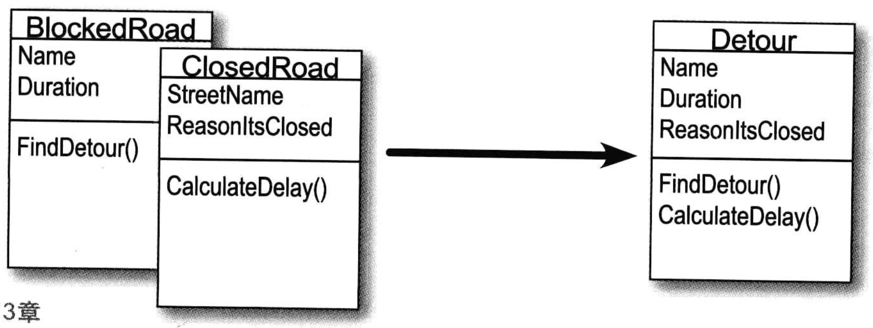

## ★ 对类和方法使用具有描述性的名字。

别人应该通过类和方法的名字就能明确它们的作用。




## ★ 查找类之间的相似性。

有时两个类如果确实非常相似，可以将它们合并为一个类。制糖系统可能有3～4个涡轮，不过只有一个关闭阀门的方法（可以取涡杆号作为参数）。



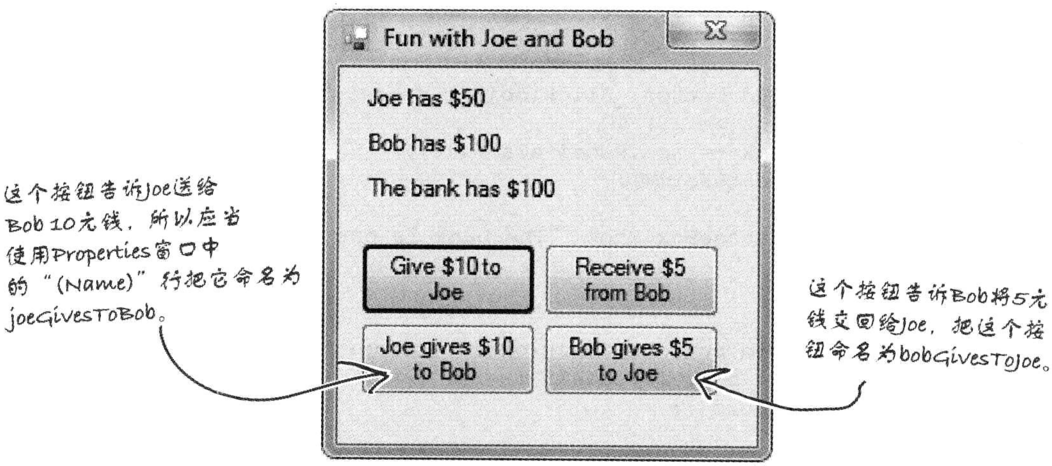
**Exercise**

向“Fun with Joe and Bob”程序增加按钮，使两个人能相互给钱。

1. **使用一个对象初始化方法初始化Bob的Guy实例。**
前面已经对Joe的Guy实例做了这个工作。下面再用对象初始化方法完成Bob实例的初始化。

如果已经单击了这个按钮，则先将其删除，再重新增加到窗体，并重命名。然后删除IDE原来增加的button3_Click()方法，而使用它现在增加的新方法。

2. **为窗体增加另外两个按钮。**
第一个按钮告诉Joe给Bob 10元钱，第二个按钮告诉Bob将5元钱交回Joe。双击按钮之前，先到Properties窗口中通过“(Name)”行（在属性表的最上面）改变每个按钮的名字。将第一个按钮命名为joeGivesToBob，第二个按钮命名为bobGivesToJoe。




3. **让按钮起作用。**
在设计工具中双击`joeGivesToBob`按钮。IDE会向窗体增加一个名为`joeGivesToBob_Click()`的方法，每次单击这个按钮时都会运行这个方法。填写这个方法，让Joe送给Bob 10元钱。然后双击另一个按钮，在IDE创建的`bobGivesToJoe_Click()`方法中填入代码，让Bob交给Joe 5元钱。确保每次现金易手之后窗体要自行更新。

## Exercise Solution

向“Fun with Joe and Bob”程序增加按钮，使两个人能相互给钱。

```
public partial class Form1 : Form {
    Guy joe;
    Guy bob;
    int bank = 100;

    public Form1() {
        InitializeComponent();
        bob = new Guy() { Cash = 100, Name = "Bob" };
        joe = new Guy() { Cash = 50, Name = "Joe" };
        UpdateForm();
    }

    public void UpdateForm() {
        joesCashLabel.Text = joe.Name + " has $" + joe.Cash;
        bobsCashLabel.Text = bob.Name + " has $" + bob.Cash;
        bankCashLabel.Text = "The bank has $" + bank;
    }

    private void button1_Click(object sender, EventArgs e) {
        if (bank >= 10) {
            bank -= joe.ReceiveCash(10);
            UpdateForm();
        } else {
            MessageBox.Show( "The bank is out of money." );
        }
    }

    private void button2_Click(object sender, EventArgs e) {
        bank += bob.GiveCash(5);
        UpdateForm();
    }

    private void joeGivesToBob_Click(object sender, EventArgs e) {
        bob.ReceiveCash(joe.GiveCash(10));
        UpdateForm();
    }

    private void bobGivesToJoe_Click(object sender, EventArgs e) {
        joe.ReceiveCash(bob.GiveCash(5));
        UpdateForm();
    }

}
```

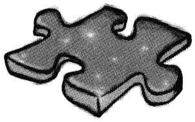
这里是Guy类两个实例的对象初始化方法。Bob初始化为有100元钱，并设置了他的名字。

为了让Joe把钱交给Bob，调用Joe的GiveCash()方法，并将其结果发送到Bob的ReceiveCash()方法。

再仔细看看Guy方法是如何调用的。GiveCash()返回的结果作为参数直接交给ReceiveCash()。

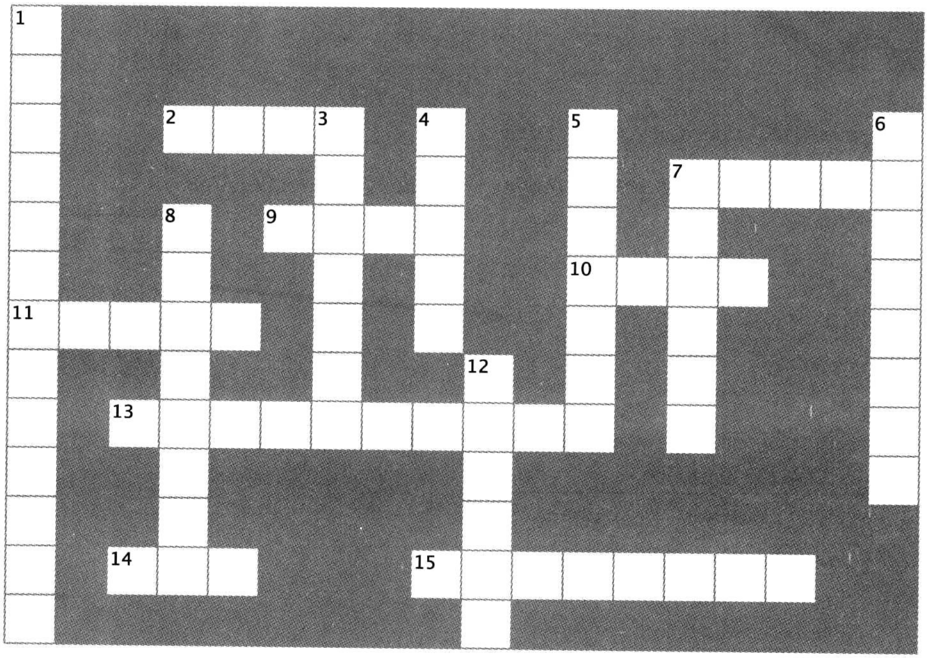
这里的难点是需要考虑谁在给钱，谁在收钱。

继续学习下面的内容之前，可以先花一点时间翻到附录“其他”，因为有一些基本语法我们还没有谈到。尽管这些内容不妨碍后面的学习，不过最好还是看一看附录里有些什么。



# 对象填字游戏

现在让你的左脑休息一下，请开动右脑；这里的所有词都与对象有关，而且都取自这一章。




## 横向

2. 如果一个方法的返回类型是 ______,则这个方法不返回任何结果。
7. 一个对象的字段定义了它的 ________。
9. 好的方法 ___________ 能清楚地表明这个方法做什么。
10. 对象存在于___________。
11. 要用___________来构建对象。
13. 要用___________将信息传入方法。
14. 使用___________语句来创建对象。
15. 用_________来设置控件和其他类的属性。

## 纵向

1. 这个窗体控件允许用户从你设置的范围中选择一个数。
3. 开始编写代码之前先在纸上创建一个类_________是一个很好的想法。
4. 对象用_________来跟踪它知道的信息。
5. __________定义了一个对象做的工作。
6. 对象的方法定义了它的 _________。
7. 如果希望创建一个类的实例，则在类声明中就不要使用这个关键字。
8. 对象是类的一个 _______________ 。
12. 这个语句告诉一个方法立即退出，并指定要传回的值（返回给调用该方法的语句）。

# 池塘谜题答案



你的任务是从这个池塘里取出一些代码片段，在代码中填空。你的目标是建立一个能够编译并运行的类，并生成所列的输出。

```
public partial class Form1 : Form
{
  private void button1_Click(object sender, EventArgs e)
  {
    String result = "";
    Echo e1 = new Echo();
    Echo e2 = new Echo();
    int x = 0;
    while ( x < 4 ) {
      result = result + e1.Hello() + "\n";
      e1.count = e1.count + 1;
      if ( x == 3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    MessageBox.Show(result + "Count: " + e2.count);
  }

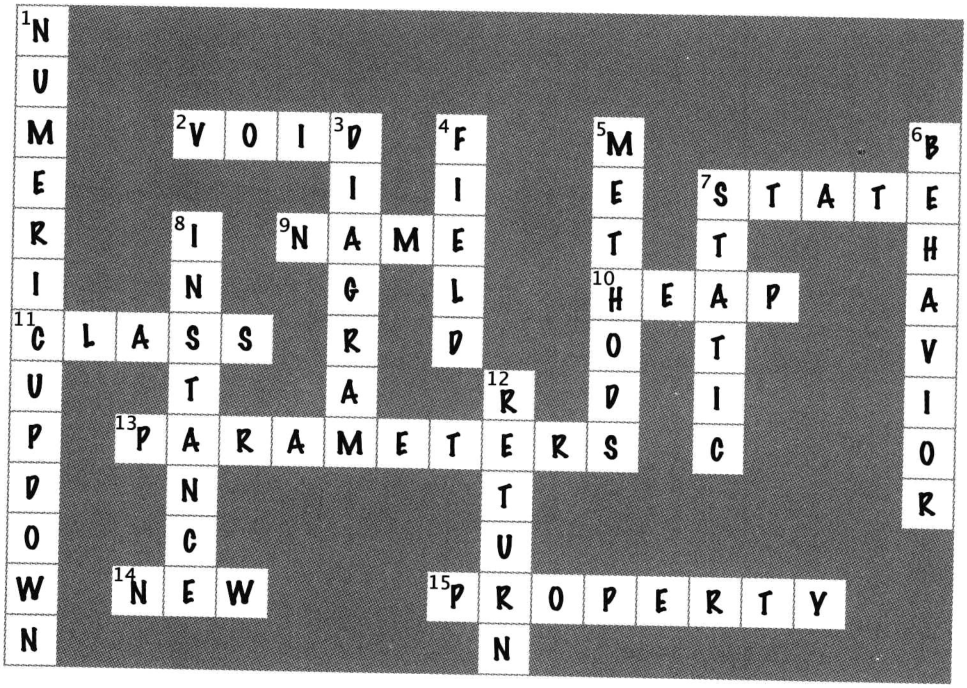
  class Echo {
    public int count = 0;
    public string Hello() {
      return "helloooo...";
    }
  }
}
```

这是正确答案。

附加问题的答案如下！

Echo e2 = e1;

# 对象填字游戏答案


1 NUMERICUPDOWN
2 VOID
3 DIAGRA
4 FIELD
5 METHODS
6 BEHAVIOR
7 STATE
8 INSTANCE
9 NAME
10 HEAP
11 CLASS
12 RETURN
13 PARAMETERS
14 NEW
15 PROPERTY
STATIC

# 4 类型与引用

# 现在是上午10:00，你知道你的数据在哪里吗？




**数据类型、数据库、军官数据……这些都很重要。**如果没有数据，你的程序毫无用处。你需要用户提供的信息，要使用这些信息来查找或产生新的信息，再交给用户。实际上，编程中所做的几乎每一件事都是在以这样或那样的方式处理数据。在这一章中，你将全面了解C#的数据类型，学习如何在程序中处理数据，甚至还会发现关于对象的一些小秘密（知道吗？对象也是数据）。

# 变量的类型决定了它能存储哪种数据

C#内置有多种类型，分别存储不同种类的数据。前面已经见过一些最常用的类型，你也知道了如何使用这些类型。不过还有很多类型你没有见过，这些类型的使用同样也很方便。

### 最常使用的值类型

毫不奇怪，最常用的类型是int、string、bool和float。

整数没有小数点。

★ int可以存储−2147483648～2147483647之间的任何整数。

★ string可以存储任意长度的文本（包括空串“”）。

★ bool是一个布尔值，可以为true或false。

★ double可以存储$\pm 5.0 \times 10^{-324}$～$\pm 1.7 \times 10^{308}$之间的任意小数（最多16位有效数字）。这个范围看上去很奇怪，也很复杂，不过实际上很简单。“有效数字”部分表示一个数的精度：35048410000000、1743059、14.43857和0.00004374155都有7位有效数字。$10^{308}$表示可以存储最大为$10^{308}$的数（1后面有308个0），只要它有16个或少于16个有效数字。来看这个范围的下界，$10^{-324}$表示可以存储最小为$10^{-324}$的数（或小数点后而且1之前有324个0）……不过，可以想见，同样要求只能有16个或少于16个有效数字。

“float”是“floating point”（浮点数）的简写，它对应于固定点数（“fixed point”），后者小数位的位数相同。

大多数情况下，如果你在使用这些类型，这是因为你所解决的问题中，可能需要一种“回转”效果，这对于问题的解决可能很有帮助，有关内容稍后就会介绍。

### 更多表示整数的类型

原先计算机内存价格相当昂贵，处理器速度也非常慢。而且，不管你相信与否，如果使用了错误的类型，有可能严重影响程序，使速度大幅下降。好在时过境迁，如今大多数情况下，用int存储整数就足够了。不过有时确实还需要一些更大的类型…… 另外有时可能还需要一些更小的类型。正因如此，C#为你提供了更多选择：

★ byte可以存储0～255的任意整数。

★ sbyte可以存储−127～128的任意整数。

★ short可以存储−32767～32768的任意整数。

★ ushort可以存储0～65535的任意整数。

★ uint可以存储 0～4294967295的任意整数。

★ long可以存储$\pm 9 \times 10^{18}$之间的任何整数。

★ ulong可以存储0～$18 \times 10^{18}$之间的任何整数。

sbyte中的“s”代表“有符号”(signed)，表示它可以是一个负数（“符号”是一个负号）。

“u”代表“无符号”(unsigned)。

## 存储非常大和非常小的数值类型

有时7位有效数字还不够精确。而且不论你是否相信，有时$10^{38}$都不够大，$10^{-45}$也不够小。为金融和科学研究领域编写的很多程序经常会遇到这些问题，所以C#还提供了另外两个类型：

★ `float`可以存储$\pm1.5\times10^{-45}\sim\pm3.4\times10^{38}$之间的任何数（有7位有效数字）。

★ `decimal`可以存储$\pm1.0\times10^{-28}\sim\pm7.9\times10^{28}$之间的任何数（有28、29位有效数字）。

程序需要处理货币时，通常会希望使用decimal来存储数值。

使用numericUpDown控件的Value属性时，就在使用一个decimal。

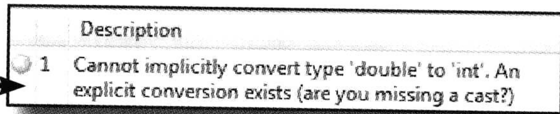
### 字面量也有类型。

所谓字面量是指在代码中输入的一个数。所以输入“int i = 5;”时，5就是一个字面量。

在C#程序中直接输入一个数时，就是在使用字面量（literal）…… 会为每个字面量自动指定一个类型。你可以自己试试看，输入下面这行代码，将字面量14.7赋给一个int变量：

```
int myInt = 14.7;
```

现在试着构建这个程序，你会得到：




如果试图设置一个int等于一个double变量，则也会得到同样的错误。这是IDE在告诉你，字面量14.7有一个类型，这个类型就是double。可以在最后加一个F（写作14.7F）将其类型改为float。另外，如果加一个M（写作14.7M）则是decimal类型。

“M”代表“money”（钱）。说正经的，确实是这样！

如果试图将float字面量赋至一个double变量或者将decimal字面量赋至一个float变量，则IDE会给出一个有用的消息，提醒你增加合适的后缀。真是很酷！

### 一些非常有用的内置类型

有时需要存储单个字符，如Q、7或$，这种情况下就可以使用char类型。char的字面量值总是放在单引号里(如‘x’和‘3’)。还可以在引号里包含转义序列（escape sequences），如‘\n’是一个换行符，‘\t’是tab制表符。在C#代码中用两个字符来写转义序列，不过程序会把各个转义序列作为单个字符存储在内存中。

第9章中你将对char和byte的关系有更多的了解。

最后还有一个更重要的类型：object。你已经看到了可以通过创建类的实例来创建对象，没错，每一个这样的对象都可以赋至一个object变量，本章后面你将充分了解对象以及引用对象的变量如何工作。

Windows 7的计算器中有一个很不错的特性，叫做“程序员”模式，采用这种模式时，你可以同时看到数值的二进制和十进制表示。

**BRAIN POWER**

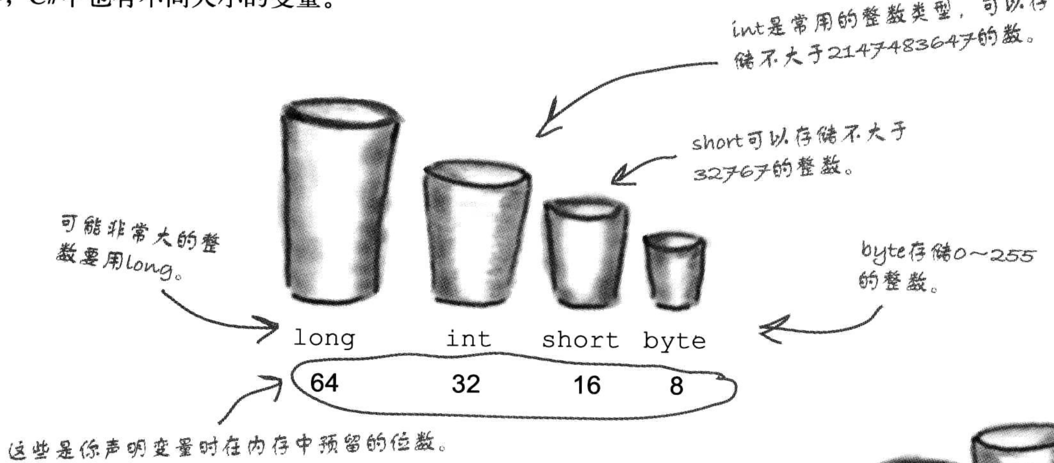
可以使用Windows计算器在十进制数（正常的以10为基数的数）和二进制（以2为基数，只由1和0组成）之间转换，采用科学型模式，输入一个数，单击Bin（二进制）单选钮，就可以把它转换为二进制。然后单击Dec（十进制）单选钮，再转换回十进制数。现在输入整数类型的上下界（如－32,767和255），把它们转换为二进制数。你能想出为什么C#会给定这些上下界吗？

# 变量就像数据外卖杯

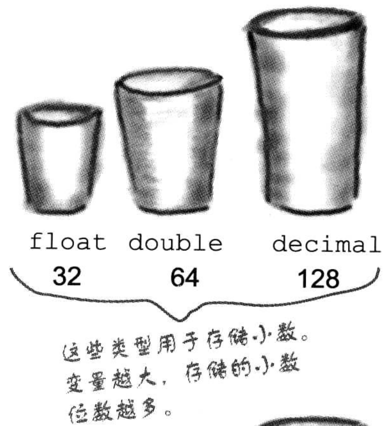
所有数据都会占据内存中的一定空间（还记得上一章中的堆吗），所以你必须考虑：程序中使用一个串或一个数时需要多少空间。这正是使用变量的原因之一。利用变量，可以在内存中预留足够的空间来存储数据。

并不是所有数据最后都存放在堆上。值类型（value types）通常将数据存储在内存中的另一部分，称为栈（stack）。有关内容将在第14章介绍。

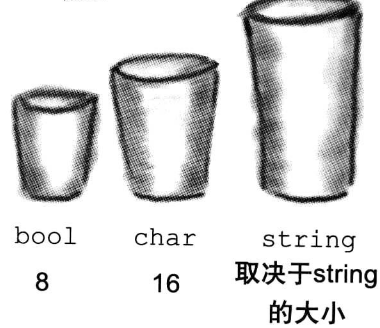
可以把变量想象成一个杯子，能够在其中放入你的数据。C#使用了一组不同的杯子来装不同的数据。就像咖啡馆里不同大小的杯子一样，C#中也有不同大小的变量。



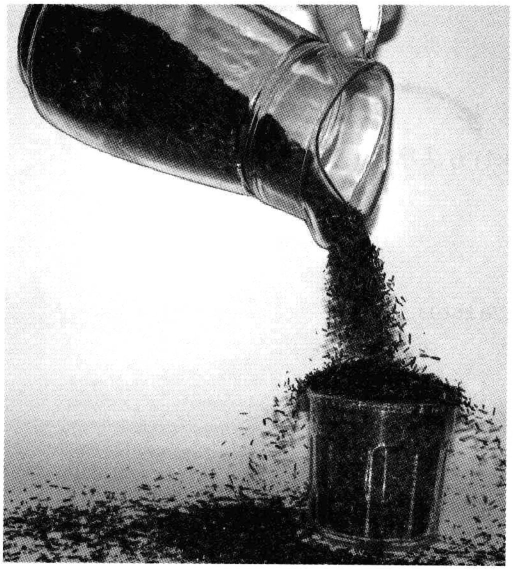

带小数的数存储时与整数有所不同。大多数带小数的数都可以使用float，这是存储小数的最小的数据类型。如果需要更精确一些，可以使用double，如果你在编写一个金融应用，需要存储货币值，则可能希望使用decimal类型。




不过，并不只是有数字而已（你肯定不希望把热咖啡装在一个塑料杯子里，也不会把冰咖啡放在一个纸杯子里）。C#编译器还能处理字符和非数值类型。char类型存储一个字符，string用于存储“串”在一起的多个字符。string对象没有固定的大小。它能扩展到足够大，保证需要在其中存储的数据确实能够放下。bool数据类型用于存储true或false值，if语句中就曾用过这种bool类型。



# 10磅数据装进5磅的包里

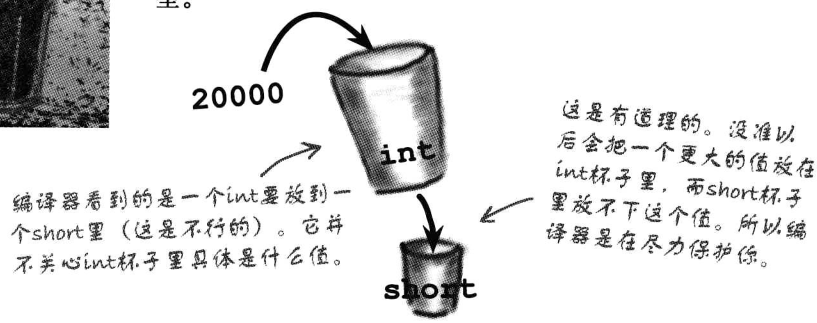


将变量声明为某种类型时，编译器就会把它看做是这种类型。即使变量值与所声明类型的上界相去甚远，编译器所看的是数据所在的杯子，而不是其中装的数据。所以下面这样做是不行的：

```
int leaguesUnderTheSea = 20000;

short smallerLeagues = leaguesUnderTheSea;
```

20000可以放在一个short里，这没有问题。不过，由于leaguesUnderTheSea声明为一个int，编译器把它看做int大小，认为它太大了，无法放在一个short容器里。编译器不会动态地为你完成这种转换。需要由你确保所处理的数据使用了适当的类型。




## Sharpen your pencil

以下有3个语句不能编译，可能是因为试图把过大的数据装进一个小变量中，或者因为所放入数据的类型不正确。把这3个语句圈出来。

```
int hours = 24;

short y = 78000;

bool isDone = yes;

short RPM = 33;

int balance = 345667 - 567;

string taunt = "your mother";

byte days = 365;

long radius = 3;

char initial = 'S';

string months = "12";
```

# 即使数字的大小合适，也不能随意赋给任意变量

下面来看如果把一个decimal值赋给一个int变量会发生什么情况。

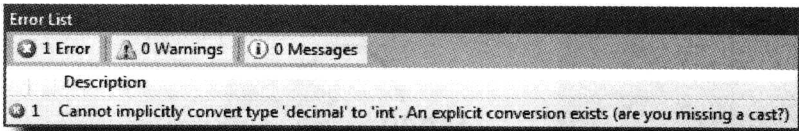
**动手做！**

❶ 创建一个新工程，并增加一个按钮。然后向按钮的Click()方法增加下面几行代码：

```
decimal myDecimalValue = 10;
int myIntValue = myDecimalValue;

MessageBox.Show("The myIntValue is " + myIntValue);
```

❷ 构建这个程序。唉呀，你会看到这样一个错误：




想想看IDE如何发现你可能缺少一个强制类型转换。

❸ 将decimal强制转换为一个int可以去除这个错误。如下修改第二行后，程序将顺利编译并运行：

```
int myIntValue = (int) myDecimalValue;
```

在这里将decimal值强制转换为一个int。

## 出了什么问题？

如果一个值的类型与变量不一致，即使这个变量确实可以存储这个值，编译器也不允许将值赋给该变量，因为类型不一致的赋值是导致大量bug的根本原因。使用强制类型转换时，实际上是在向编译器做出一个承诺，表明你知道它们的类型不一样，但在这个特定情况下C#确实可以把数据填到新变量里。

先花一点时间翻回到上一章最前面，看看向Talker Tester窗体传入NumericUpDown值时是如何使用强制类型转换的。

## Sharpen your pencil Solution

其中3个语句不能编译，可能是因为它们试图把过大的数据装进一个小变量中，或者是因为所放入数据的类型不正确。把它们圈出来。

```
short y = 78000;
```

short类型只能存储-32767～32768之间的数。这个数太大了！

```
byte days = 365;
```

byte只能保存最大为256的值。这里需要一个short。

```
bool isDone = yes;
```

只能向bool赋一个"true"或"false"值。

# 对一个过大的值完成强制类型转换时，C#会自动进行调整

你已经看到了，decimal可以强制转换为一个int。可以明确，任何数都可以强制转换为另一个数。不过，这并不是说在强制类型转换过程中这个值原封不动。如果将一个设置为365的int变量强制转换为一个byte变量，对于byte来说，365太大了。不过并不会因此显示一个错误，而只是把这个值回转（wrap around），例如，256强制转换为一个byte时，将得到值0。257将转换为1，258转换后会得到2，依此类推直到365，最后这会得到109。一旦再次回到255时，转换值会再次“回转”到0。




**没错！+操作符在为你完成强制类型转换。**

你所做的只是使用+操作符，它会自动为你做大量强制类型转换工作，不过它在这方面相当聪明。使用+把一个数字或布尔值加至一个串时，它会自动地把这个值转换为串。如果对两个不同的类型使用+（或*，/或-），则它会自动地把较小的类型转换为较大的类型。以下是一个例子：

```
int myInt = 36;
double myFloat = 16.4D;
myFloat = myInt + myFloat;
```

将一个数值赋至double时，需要在这个数字后面加一个D，告诉编译器这是一个double而不是float。

由于int可以放入float中，而float无法放入int中，所以+操作符将myInt与myFloat相加之前，会先把myInt强制转换为float。

## 自己动手！

强制类型转换如何完成数字的“回转”并不神秘，你自己也可以做到。只需打开Windows计算器，切换到科学型模式，计算365 Mod 256（用“Mod”按钮，这会完成一个取余计算），你就会得到109。

## Sharpen your pencil

并不是任何类型都能强制转换为任何其他类型。创建一个新工程，向窗体拖入一个按钮，双击按钮，然后在这个按钮的方法中输入以下语句。构建这个程序时，会显示很多错误。去掉那些导致错误的语句。这能帮助你搞清楚哪些类型可以完成强制转换，而哪些不能！

```
int myInt = 10;

byte myByte = (byte)myInt;

double myDouble = (double)myByte;

bool myBool = (bool)myDouble;

string myString = "false";

myBool = (bool)myString;

myString = (string)myInt;

myString = myInt.ToString();

myBool = (bool)myByte;

myByte = (byte)myBool;

short myShort = (short)myInt;

char myChar = 'x';

myString = (string)myChar;

long myLong = (long)myInt;

decimal myDecimal = (decimal)myLong;

myString = myString + myInt + myByte
+ myDouble + myChar;
```

# C#自动完成一些强制类型转换

有两个不需要你强制完成的重要转换。首先，使用算术操作符时会自动完成转换，如下面这个例子：

```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
MessageBox.Show("The answer is " + d);
```

-操作符从long减去short，=操作符把结果转换为一个double。

+操作符相当聪明，它会把decimal转换为一个string。

使用+操作符连接（concatenate）串时（这表示将一个串追加到另一个串的末尾，前面生成消息框时已经这样做过），C#会自动完成另一种转换。使用+将一个串与另一种类型的某个值或变量连接时，它会自动地将数字转换为串。下面是一个例子。前面两行没问题，不过第三行将无法编译。

```
long x = 139401930;
MessageBox.Show("The answer is" + x);
MessageBox.Show(x);
```

C#编译器会产生一个错误，提示有非法实参[C#中将传入方法参数（也称为行参，parameter）的值称为实参(argument)]。这是因为MessageBox.Show()的参数是一个string，而这个代码传入了一个long，对于这个方法来说这是一个错误的类型。不过，可以很容易地把它转换为一个串，只需调用其ToString()方法。所有值类型和对象都有这个方法（你自己构建的所有类也有一个ToString()方法，它会返回类名）。可以如下将x转换为MessageBox.Show()可以使用的串：

```
MessageBox.Show(x.ToString());
```

## Sharpen your pencil Solution

并不是任何类型都能强制转换为任何其他类型。创建一个新工程，向窗体拖入一个按钮，双击按钮，然后在这个按钮的方法中输入以下语句。构建这个程序时，会显示很多错误。去掉那些导致错误的语句。这能帮助你搞清楚哪些类型可以完成强制转换，而哪些不能！

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;            (struck through)
string myString = "false";
myBool = (bool)myString;                 (struck through)
myString = (string)myInt;                (struck through)
myString = myInt.ToString();
myBool = (bool)myByte;                   (struck through)
myByte = (byte)myBool;                   (struck through)
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;               (struck through)
long myLong = (long)myInt;
decimal myDecimal = (decimal)myLong;
myString = myString + myInt + myByte
+ myDouble + myChar;
```

## 调用一个方法时，实参必须与行参的类型兼容

行参（parameter）是方法中定义的参数。实参（argument）是向方法传入的参数。如果一个方法有一个int行参，它可以接受一个byte实参。

试着调用MessageBox.Show(123)，也就是向MessageBox.Show()传入一个字面量(123)而不是一个串。IDE将不允许构建这个程序。实际上，IDE中会显示一个错误：“Argument ‘1’: cannot convert from ‘int’ to ‘string’.”（实参‘1’：无法从‘int’转换为string’）。

**编译器给出一个“非法参数”错误时，这说明你试图调用一个方法，而你提供的变量与该方法的参数类型不匹配。**

不过并不只是MessageBox.Show()在这种情况下会给出编译错误，对于其他方法，如果为方法传入的变量与参数的类型不匹配，也会产生这种错误。所有方法都是如此，甚至包括你自己写的方法。下面再在一个类中输入以下这个完全合法的方法:

```
public int MyMethod(bool yesNo) {

   if (yesNo) {
      return 45;
   } else {
      return 61;
   }

}
```

提醒一句，调用这个方法的代码并不一定非得传入一个名为yesNo的变量，只需传入一个布尔值或布尔变量就可以。这个参数只是在该方法代码中才名为yesNo。

如果传入方法期望的值（一个bool），则这个方法能很好地工作，如果调用MyMethod(true)或MyMethod(false)，则它将正常编译。

可以为类型为object的变量、参数或字段赋任何值。

但是如果传入一个整数或一个串呢？IDE会给出一个错误，就像前面为MessageBox.Show()传入123一样，你会看到一个类似的错误。下面试着传入一个布尔值，不过将返回值赋给一个string，或者将返回值传入MessageBox.Show()。这也无法正常工作，方法会返回一个int，而不是long或者MessageBox.Show()所期望的string。

### if语句总是查看是否为true

请注意我们是这样写if语句的：

if (yesNo) {

这里没有必要明确地指出“if (yesNo == true)”。这是因为，if语句总是查看某个值是否为true。要检查是否为false可以使用！[感叹号，或非（NOT）操作符]。“if (!yesNo)”完全等同于“if (yesNo == false)”。从现在起，在我们的代码示例中，你会经常看到只用“if (yesNo)”或“if (!yesNo)”，也不会明确地检查一个布尔量是否为true或false。




实际上，C#提供了一种途径可以将保留关键字用作为变量名，只需在关键字前面加@。如果你愿意，则可以对非保留名这样做。

C#中有大约77个保留字。它们由C#编译器所保留，不能用做变量名。读完本书后你就会了解所有这些保留字。以下给出前面已经用到的一些保留字。你认为这些保留字在C#中起什么作用？请写下来。

**namespace**

**for**

**class**

**public**

**else**

**new**

**using**

**if**

**while**

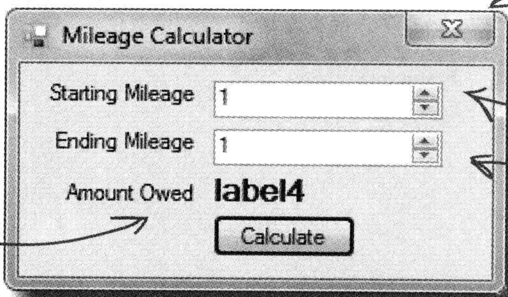
答案见164页。

**Exercise**

为出差报销创建一个报销计算器。允许用户输入一个开始里程数和一个结束里程数。根据这两个数，可以计算出行程多少英里，并得出应该给用户报销多少钱（假设每1英里行程公司给用户报销0.39美元）。

**1 首先创建一个新的Windows工程。**

窗体如下：




完成窗体后，双击按钮，为工程增加一些代码。

**2 创建计算器所需的变量。**

把变量放在Form1最上面的类定义中。这里需要两个整数变量来记录开始里程数和结束里程数。可以把这两个变量命名为startingMileage和endingMileage。还需要3个能存储小数的数，指定类型为double，并分别命名为milesTraveled、reimburseRate和amountOwed。将reimburseRate的值设置为.39。

**3 完成这个计算器。**

在button1 _ Click()方法中增加代码：

- 确保Starting Mileage输入域中的数小于Ending Mileage域中的数。如果不是这样，显示一个消息框，指出“The starting mileage must be less than the ending mileage”（开始里程数必须小于结束里程数）。将这个消息框的标题置为“Cannot Calculate”（无法计算）。
- 利用以下代码行，从结束里程数减去开始里程数，然后将结果乘以报销费率：

```
milesTraveled = endingMileage -= startingMileage;
amountOwed = milesTraveled *= reimburseRate;
label4.Text = "$" + amountOwed;
```

**4 运行程序。**

确保它能给出正确的得数。试着把开始里程数改为大于结束里程值，保证它能给出以上消息框。

**Exercise Solution**

要求你为出差报销创建一个报销计算器。以下是这个练习第一部分的代码。

```
public partial class Form1 : Form
{
    int startingMileage;
    int endingMileage;
    double milesTraveled;
    double reimburseRate = .39;
    double amountOwed;
    public Form1() {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e){
      startingMileage = (int) numericUpDown1.Value;
      endingMileage = (int)numericUpDown2.Value;
        if (startingMileage <= endingMileage){
            milesTraveled = endingMileage -= startingMileage;
            amountOwed = milesTraveled *= reimburseRate;
            label4.Text = "$" + amountOwed;
        } else {
            MessageBox.Show(
    "The starting mileage must be less than the ending mileage",
                         "Cannot Calculate Mileage");
        }
    }
}
```

int对于存储整数很合适。这个数最大可以达到999,999。所以short或byte不合适。

还记得吗？必须将numericUpDown控件的结果从decimal值转换为int。

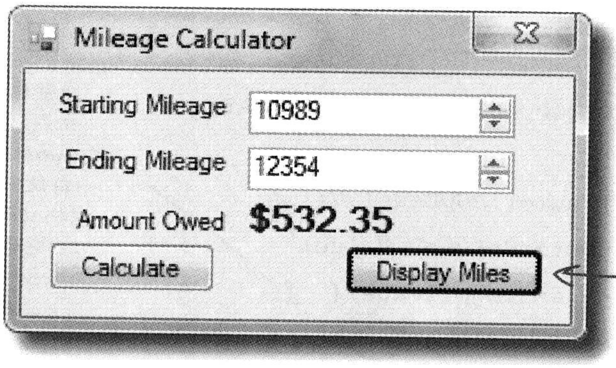
这个代码块要计算出行程英里数，然后将结果乘以报销费率。

这里我们采用了另外一种方式来调用MessageBox.Show()方法。在此指定了两个参数：第一个是要显示的消息，第二个参数是标题栏中的文本。

**这个按钮看上去能正常工作，不过这里有一个非常严重的问题，你能发现这个问题吗？**

**1** **现在再为窗体增加一个按钮。**

为了跟踪这个问题，在窗体上增加一个按钮来显示milesTraveled字段的值（也可以使用调试工具）。




完成窗体后，双击Display Miles按钮，为工程增加一些代码。

**2** **只需一行代码。**

我们所要做的就是让窗体显示milesTraveled变量，对不对？所以只需下面这一行代码：

```
private void button2_Click(object sender, EventArgs e) {
        Messagebox.Show(milesTraveled + " miles", "Miles Traveled");
}
```

**3** **运行程序。**

输入一些值，看看会得到什么结果。首先输入一个开始里程数和一个结束里程数，单击Calculate按钮。再单击Display Miles按钮来查看milesTraveled字段中存储的值。




**4** **嗯，有点问题……**

不论使用什么数，里程数总是与报销数额一样，为什么？

# 结合=和操作符

前面已经将结束里程数减去开始里程数，请仔细看看当时我们使用的操作符(-=)。问题就在于，这个操作符不仅仅完成减法，它还会把一个值赋给减号左边的变量。后面将里程数乘以报销费率时也存在同样的问题。应当将 -= 和 *= 换成- 和*：

```
private void button1_Click(object sender, EventArgs e)
{
  startingMileage = (int) numericUpDown1.Value;
  endingMileage = (int)numericUpDown2.Value;
    if (startingMileage <= endingMileage){
       milesTraveled = endingMileage -= startingMileage;
       amountOwed = milesTraveled *= reimburseRate;
       label4.Text = "$" + amountOwed;
    } else {
       MessageBox.Show("The starting mileage number must
                       be less than the ending mileage number",
                       "Cannot Calculate Mileage");
    }
```

这些称为复合操作符（compound operators）。这个操作符从endingMileage减去startingMileage，同时还会把这个新值赋给endingMileage和milesTraveled。

这样好多了——现在代码不会再改变endingMileage和milesTraveled了。

```
milesTraveled = endingMileage - startingMileage;
amountOwed = milesTraveled * reimburseRate;
```

**好变量名有没有帮助？**当然了！仔细看看每个变量用来做什么。从milesTraveled这个名字你应该已经得到了很多线索。你知道正是这个变量在窗体上未能正确地显示，而且你很清楚应该怎样计算这个值。所以检查代码查找bug时可以充分利用这些线索。如果出错的代码行如下，要找出问题可就困难得多了：

```
mT = eM -= sM;
aO = mT *= rR;
```

如果像这样对变量命名，就基本无法起到指示这些变量的作用了。

# 对象也使用变量

到目前为止，我们都将object与其他类型区别对待。不过object也是一种数据类型。就像处理数值、串和布尔值一样，代码会以同样的方式处理对象，即使用变量来处理：

## 使用int

① 写一个语句声明一个整数。

```
int myInt;
```

② 为这个新变量赋一个值。

```
myInt = 3761;
```

③ 在代码中使用这个整数。

```
while (i < myInt) {
```

## 使用对象

❶ 写一个语句声明一个对象。

```
Dog spot;
```

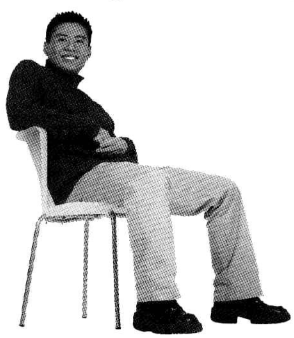
如果有一个类（如Dog），则在变量声明语句中可以将这个类用作为类型。

❷ 为这个对象赋一个值。

```
spot = new Dog();
```

❸ 检查对象的某个字段。

```
while (spot.IsHappy) {
```




**对象只是程序可以使用的另外一种类型的变量。**

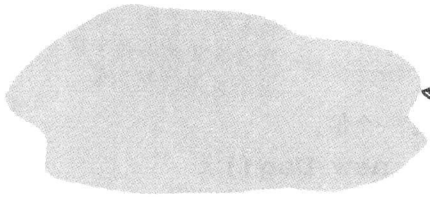
如果程序需要处理一个相当大的整数，则可以使用long。如果需要一个很小的整数，则使用short。如果需要一个yes/no值，则可以使用boolean。如果需要一个能叫能坐的东西，那就使用一个Dog。不论程序需要处理何种类型的数据，都会使用变量。

# 用引用变量指示对象

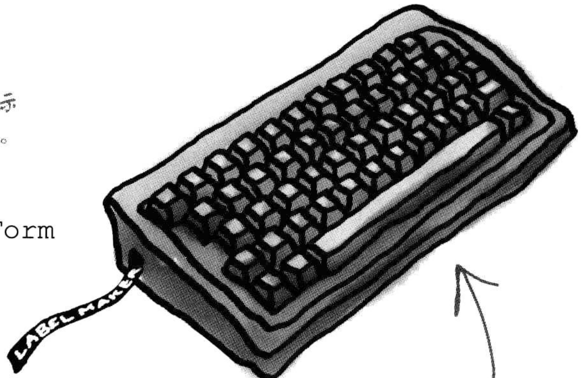
创建一个新对象时，可以使用new Guy()之类的代码。不过这还不够，尽管这个代码会在堆中创建一个新的Guy对象，但没有提供任何途径来访问这个对象。还需要对象的一个引用，所以要创建一个引用变量（reference variable）：这是一个Guy类型的变量，并有一个变量名（如joe）。所以joe就是新创建的这个Guy对象的一个引用。只要你想使用这个特定的人（“Guy”），就可以用名为joe的引用变量来引用该对象。

这称为对象的实例化。

所以，如果有一个object类型的变量，就是一个引用变量，这是指向一个特定对象的引用。来看下面的例子：



这是代码运行之前堆的示意图。现在什么也没有。




```
public partial class Form1 : Form
{
    Guy joe;

    public Form1()
    {
        InitializeComponent();

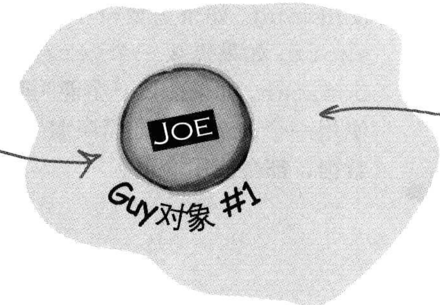
        joe = new Guy();
    }
```

这个变量名为joe，将引用一个Guy类型的对象。

这就是引用变量……

……这是joe现在引用的对象。

创建一个引用就像用标签机做一个标签，不过不是把它贴在某个东西上，而是用它来标示一个对象，以便以后引用这个对象。



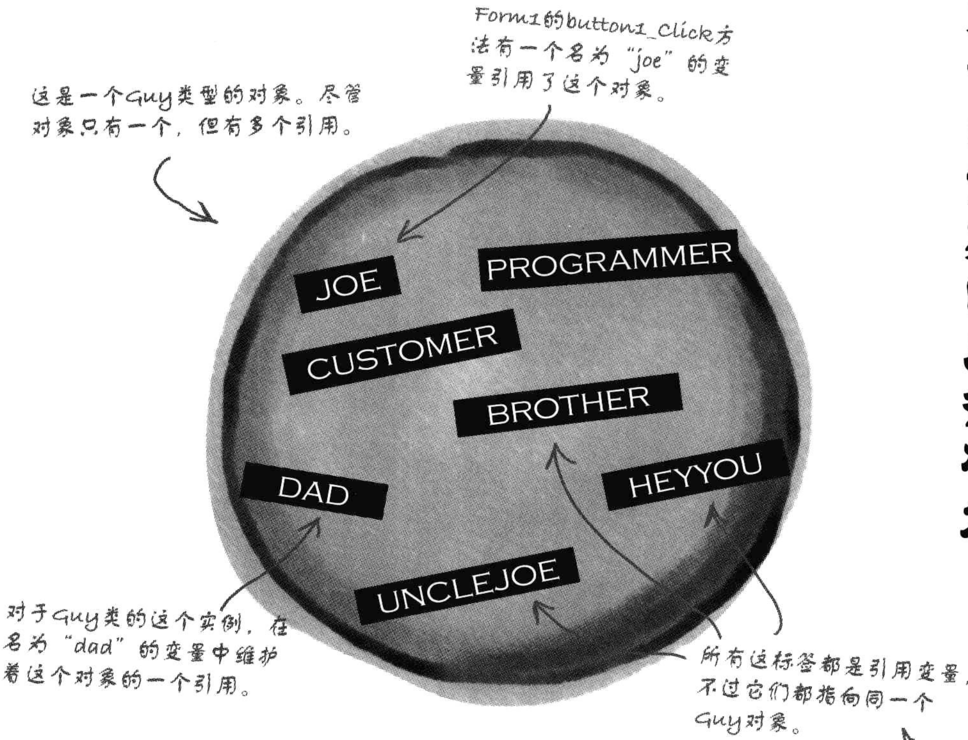

这是代码运行后堆的示意图。现在有了一个对象，由变量joe指示。

引用这个Guy对象的唯一途径就是通过名为joe的引用变量。

# 引用就像对象的标签

在你的厨房里，可能有装糖和盐的小罐子。如果把它们的标签换了，则做出来的菜可能让人难以下咽，因为尽管标签变了，但是罐子里装的东西没有变。引用就像是标签。可以移动标签，指示不同的东西，但是要由对象表明哪些方法和数据可用，而不是引用本身。

**代码需要处理内存中的一个对象时，它会使用一个引用（reference），这是一个变量，其类型是所指向的对象的类。引用就像是一个标签（label），代码用它与一个特定的对象交互。**




不能直接使用对象。例如，如果**Guy**是对象类型，不能写这样的代码：**Guy.GiveCash()**。C#编译器不知道你说的是哪一个**Guy**，因为在堆里可能有多个**Guy**实例。所以需要一个引用变量，如**joe**，为它指定一个特定的实例，如**Guy joe = new Guy()**。

现在可以调用方法了［如**joe.GiveCash()**］。**joe**指示**Guy**类的一个特定实例，而且C#编译器知道要用哪一个实例。另外，前面已经看到，可以有多个标签指向同一个实例。所以可以设置**Guy dad = joe**，然后调用**dad.GiveCash()**。这也是完全可以的（joe的孩子每天都会这么做）。

这个Guy有很多不同的引用，因为很多不同的方法用它来完成不同的工作。每个引用都有一个在不同上下文中有特殊意义的不同的名字。

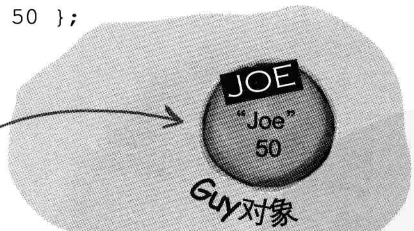
## 如果不再有任何引用，对象就会被垃圾回收

如果把一个对象的所有标签都取掉，那么程序就再也无法访问这个对象了。这说明，C#可以将对象标志为进行垃圾回收（garbage collection）。此时，C#会撤销所有没有任何引用的对象，并回收这些对象原来占用的内存以供程序以后使用。

**要让一个对象留在堆中，它必须被引用。如果对象的最后一个引用没有了，对象也将消失。**

1. **以下是创建一个对象的代码：**

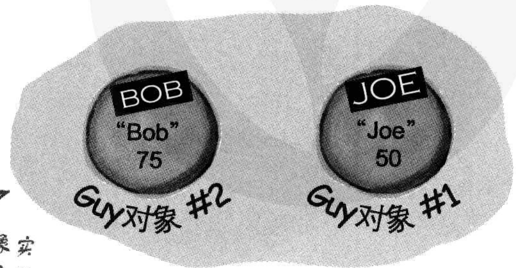
```
Guy joe = new Guy()
 { Name = "Joe", Cash = 50 };
```




使用"new"语句时，就是在告诉C#创建一个对象。将一个引用变量（如"joe"）赋值为一个对象时，就像是为它贴上一个新标签。

2. **下面创建第二个对象。**

```
Guy bob = new Guy()
 { Name = "Bob", Cash = 75 };
```



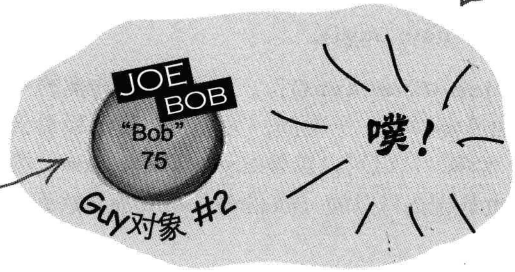

现在我们有了两个Guy对象实例，还有两个引用变量：分别指向这两个Guy实例。

3. **下面取指向第一个对象的引用，把它改为指向第二个对象。**

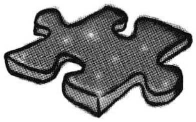
```
joe = bob;
```

现在joe与bob指向同一个对象。

不过再没有引用指向第一个Guy对象了……



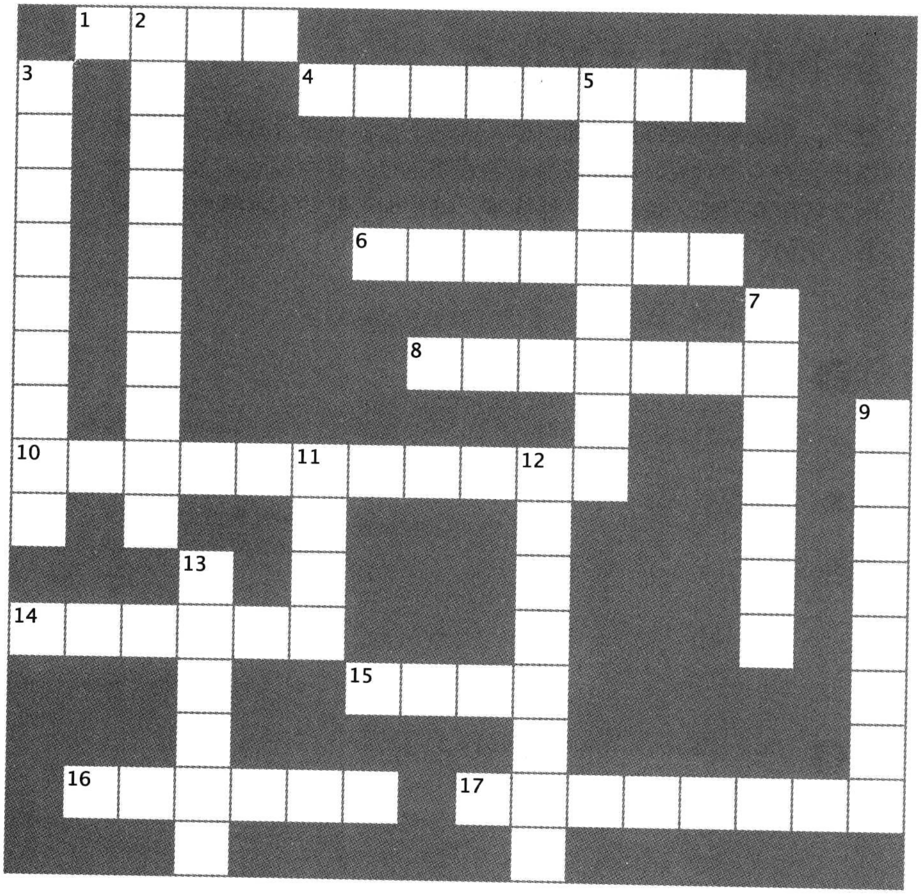

……所以C#将这个对象标志为垃圾回收。这个对象就此消失！



# 类型填字游戏

休息一下，坐下来让右脑活动活动。这是一个标准的填字游戏，所有答案都可以在这一章找到。

完成后，翻到下一页，读完这一章。



## 横向

1.变量声明的第二部分。
4.“namespace”，“for”，“while”，“using” 和“new”都是 ______________ 字的例子。
6. 在“x = (int) y;”代码行中(int)完成的工作。
8.如果再没有任何引用指向一个对象，将使用______________ 回收将其从堆中删除。
10. 使用 + 操作符将两个串连在一起时所做的工作。
14. 这种类型能存储最大的数。
15. 这种类型存储一个字母或数字。
16. \n和\r是____________序列。
17. 只能存储正数的4种整数类型。

## 纵向

2.可以把变量声明和 ______________ 合并为一个语句。
3.指向一个对象的变量。
5. 程序用______________处理内存中的数据。
7. 如果想存储一个货币值，则可以使用这种类型。
9. +=和-=都是这种操作符。
11. 变量声明总是以______________开头。
12. 每个对象都有______________方法将其转换为一个串。
13. 如果有一个这种类型的变量，则可以为它赋任何值。

答案见165页。

# 多个引用及其副作用

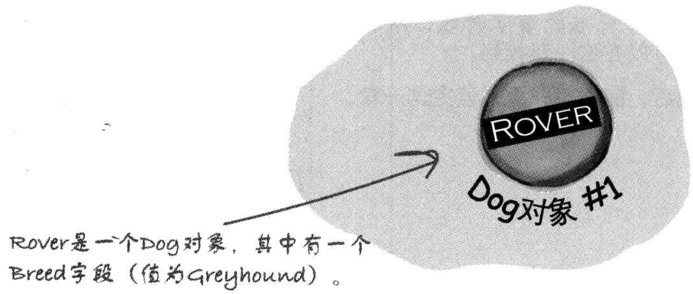
移动引用变量时要特别仔细。很多情况下，你可能只是将一个变量指向另外一个对象，不过最后却导致在这个过程中删除了另外一个对象的所有引用。这不是一件坏事，但是你可能并不希望如此。来看一个例子：

**1**

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```

对象：1

引用：1

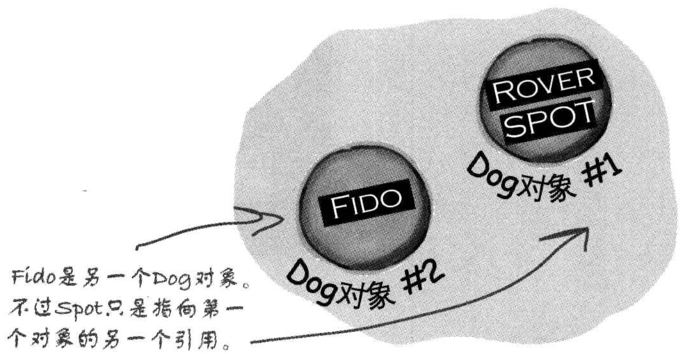



**2**

```
Dog fido = new Dog();
fido.Breed = "Beagle";
Dog spot = rover;
```

对象：2

引用：3

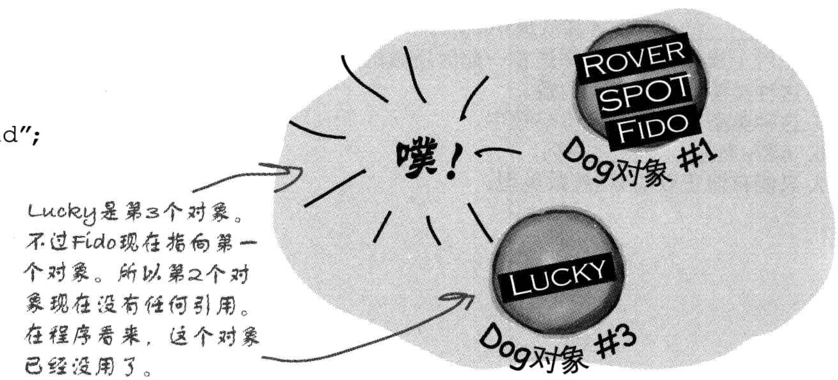



**3**

```
Dog lucky = new Dog();
lucky.Breed = "Dachshund";
fido = rover;
```

对象：2

引用：4



## Sharpen your pencil

现在轮到你了。这里有一个很长的代码块。看看每个阶段有多少个对象和引用。在右边画出堆中对象和标签的示意图。

1.
```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

**对象**：______

**引用**：_____

2.
```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

**对象**：______

**引用**：_____

3.
```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

**对象**：______

**引用**：_____

4.
```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

**对象**：______

**引用**：_____

5.
```
charlie = laverne;
lucky = rinTinTin;
```

**对象**：______

**引用**：_____

## Sharpen your pencil Solution

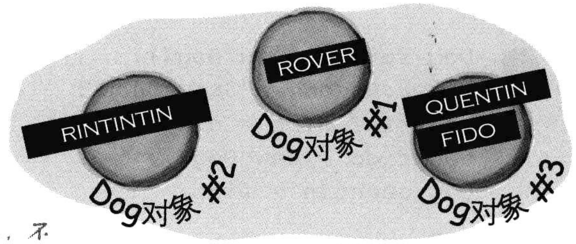
现在轮到你了。这里有一个很长的代码块。看看每个阶段有多少个对象和引用。在右边画出堆中对象和标签的示意图。

**1**

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog quentin = fido;
```

对象：3

引用：4



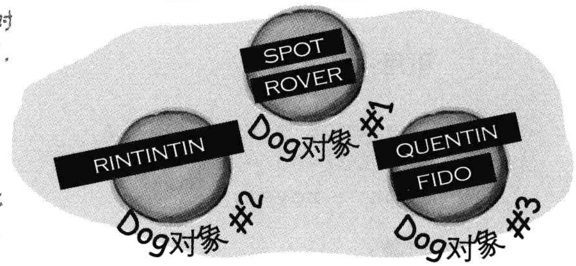

创建了一个新的Dog对象，不过只有一个引用Spot指向该对象。Spot设置为等于Rover时，这个对象就消失了。

**2**

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

对象：3

引用：5




这里创建了一个新的Dog对象，不过Fido设置为Rover时，Fido从第3个对象移出到第1个对象。

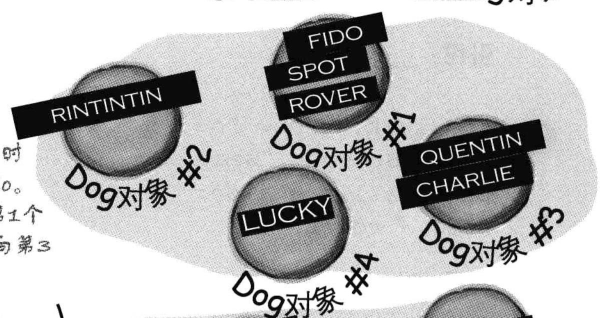
**3**

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

对象：4

引用：7

Fido还在第3个对象上时Charlie设置为等于Fido。在此之后，Fido移到第1个对象，而Charlie仍指向第3个对象。




**4**

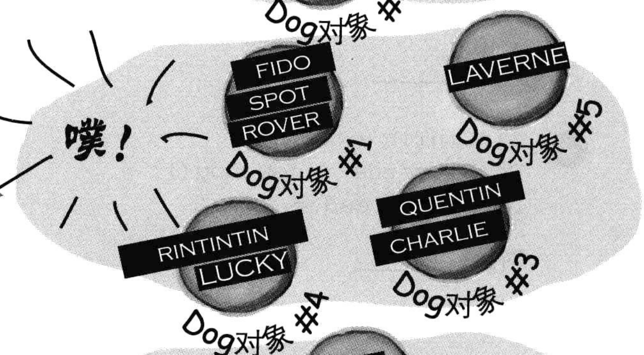
```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

对象：4

引用：8

Dog #2失去了最后一个引用，然后不见了。

Rin Tin Tin移到Lucky的对象上时，原来的Rin Tin Tin对象就消失了。



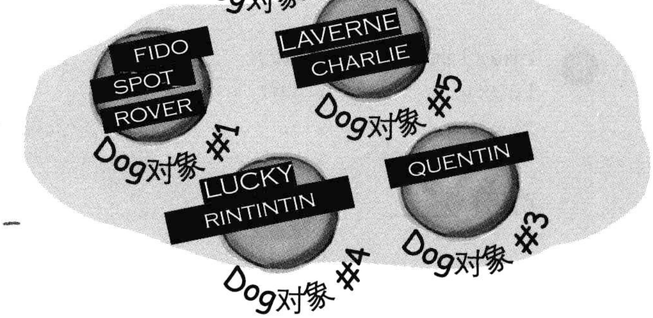

**5**

```
charlie = laverne;
lucky = rinTinTin;
```

对象：4

引用：8

这里有引用移动，但是没有创建新对象。将Lucky设置为Rin Tin Tin没有任何作用，因为它们本来就指向同一个对象。

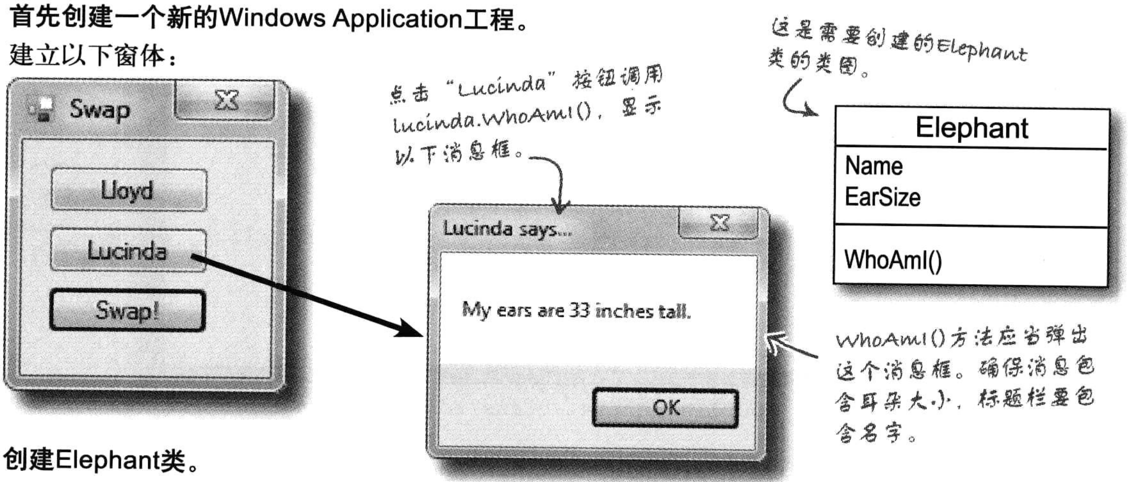





创建一个程序，包含一个大象类。建立两个大象实例，然后让它们交换属性，但不要让任何Elephant实例被垃圾回收。

**1** **首先创建一个新的Windows Application工程。**

建立以下窗体：




**2** **创建Elephant类。**

为工程增加一个`Elephant`类。先看看`Elephant`类的类图，需要一个名为`EarSize`的`int`字段，另外需要一个名为`Name`的`String`字段（要保证这两个字段都是公共的）。然后增加一个名为`WhoAmI()`的方法，显示一个消息框，指出这个大象的名字和耳朵大小。

**3** **创建两个Elephant实例和一个引用。**

为`Form1`类增加两个`Elephant`字段（在类声明下面），分别名为`Lloyd`和`Lucinda`。初始化这两个实例，使它们有正确的名字和耳朵大小。为窗体增加的**`Elephant`**对象初始化方法如下：

```
lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

**4** **让“Lloyd”和“Lucinda”按钮起作用。**

让Lloyd按钮调用`lloyd.WhoAmI()`，而让Lucinda按钮调用`lucinda.WhoAmI()`。

**5** **完成swap按钮。**

这是最难的一部分。让Swap按钮交换两个引用，单击Swap按钮时，`Lloyd`和`Lucinda`变量会交换对象，并显示一个“Objects swapped”消息框。测试你的程序，单击Swap按钮，然后单击另外两个按钮。第一次单击Swap时，Lloyd按钮应当弹出Lucinda的消息框，而单击Lucinda按钮应当弹出Lloyd的消息框。如果再次单击Swap按钮，一切都将还原。

**C#将把所有没有任何引用的对象垃圾回收。所以给你一个提示：如果你想把一杯啤酒倒入一个已经装满水的杯子，那么你还需要第三个杯子，先把水倒进那个杯子里……**

## Exercise Solution

创建一个程序，包含一个大象类。建立两个大象实例，然后让它们交换属性，但不要让任何Elephant实例被垃圾回收。

```
using System.Windows.Forms;

class Elephant {

  public int EarSize;
  public string Name;

  public void WhoAmI() {
     MessageBox.Show("My ears are " + EarSize + " inches tall.",
        Name + " says…");
  }
}
```

这是为工程增加的Elephant.cs文件中的Elephant类声明代码。不要忘记在类最前面加上"using System.Windows.Forms;"。如果没有这行代码，MessageBox语句将不能工作。

这是Form1.cs中的Form1类代码。

```
public partial class Form1 : Form {

    Elephant lucinda;
    Elephant lloyd;

    public Form1()
    {
        InitializeComponent();
        lucinda = new Elephant()
             { Name = "Lucinda", EarSize = 33 };
        lloyd = new Elephant()
             { Name = "Lloyd", EarSize = 40 };
    }

    private void button1_Click(object sender, EventArgs e) {
        lloyd.WhoAmI();
    }

    private void button2_Click(object sender, EventArgs e) {
        lucinda.WhoAmI();
    }

    private void button3_Click(object sender, EventArgs e) {
        Elephant holder;
        holder = lloyd;
        lloyd = lucinda;
        lucinda = holder;
        MessageBox.Show("Objects swapped");
    }
}
```

如果只是让Lloyd指向Lucinda，将没有任何引用指向Lloyd，它的对象就会消失。正是因为这个原因，需要有一个Holder引用保留在Lloyd对象上，直到Lucinda引用这个对象为止。

不需要new语句来建立引用，因为我们不希望再创建Elephant的另一个实例。

串和数组与之前见过的所有其他数据类型都不同，因为只有它们没有固定的大小。（请多想一想）。

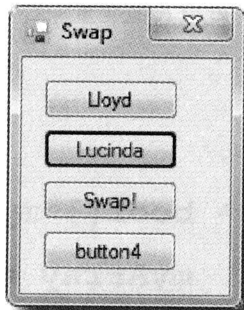
## BRAIN POWER

我们没有为Elephant类增加swap()方法，你认为这是为什么？

# 两个引用意味着有两个途径来改变一个对象的数据

前面谈到有可能丢掉一个对象的所有引用（而导致垃圾回收），除了这个问题以外，如果一个对象有多个引用，还有可能无意地修改一个对象。换句话说，对象的一个引用可能在修改对象，而该对象的另一个引用并不知道已经发生了改变。请看一个例子：

动手做！




1. 向窗体再增加一个按钮。

2. 为按钮增加以下代码。你能猜出单击这个按钮时会发生什么吗？

```
private void button4_Click(object sender, EventArgs e)
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
```

这个语句指出，将lloyd引用指向的对象的EarSize设置为4321。

运行这个代码后，lloyd和lucinda变量都会引用同一个Elephant对象。

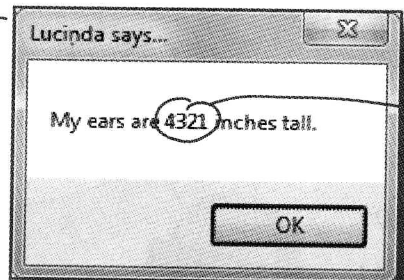
这里在从lloyd对象调用WhoAmI()方法。

不过lloyd指向的对象与lucinda恰好一样。




3. 好了，接下来单击这个新按钮。等一会，居然弹出Lucinda的消息框。难道我们调用的不是Lloyd的WhoAmI()方法吗？

这是lucinda的消息框……




却使用lloyd引用设置了这个EarSize！怎么回事？




lloyd和lucinda现在是可以互换的。对一个引用的修改会影响这两个引用所指向的对象……lloyd和lucinda之间不再有任何区别，因为它们都指向同一个对象。

注意数据没有被覆盖，改变的只是引用。

# 一种特殊情况：数组

如果需要跟踪相同类型的大量数据，比如说，一个高度列表或一群狗，可以用数组（array）来实现。数组之所以特殊，是因为这是一组变量，却处理为一个对象。利用数组，可以存储和修改多个数据，而不必单独地跟踪每一个变量。创建一个数组时，就像其他变量一样，要用一个名和一个类型来声明：

```
bool[] myArray;
myArray = new bool[15];
myArray[4] = true;
```

声明数组时，指定其类型，后面是中括号。

使用new关键字创建数组，因为数组也是一个对象。所以数组变量就是一种引用变量。

可以将myArray变量的声明与它的初始化结合在一起，就像任何其他变量一样。这就会得到：

bool[] myArray = new bool[15];

这个数组中有15个元素。

这行代码设置myArray的第5个元素值为true。之所以是第5个，这是因为第一个元素是myArray[0]，第二个元素是myArray[1]，依此类推。

## 数组中单个元素的使用类似于一个正常的变量

使用数组时，首先需要声明一个指向这个数组的引用变量。然后需要使用new语句创建数组对象，并指定数组的大小。然后可以设置这个数组中的元素。下面是一个代码示例，其中声明并填充了一个数组，执行这个代码时，堆中会发生什么变化？要记住，数组中第一个元素的索引（index）为0。

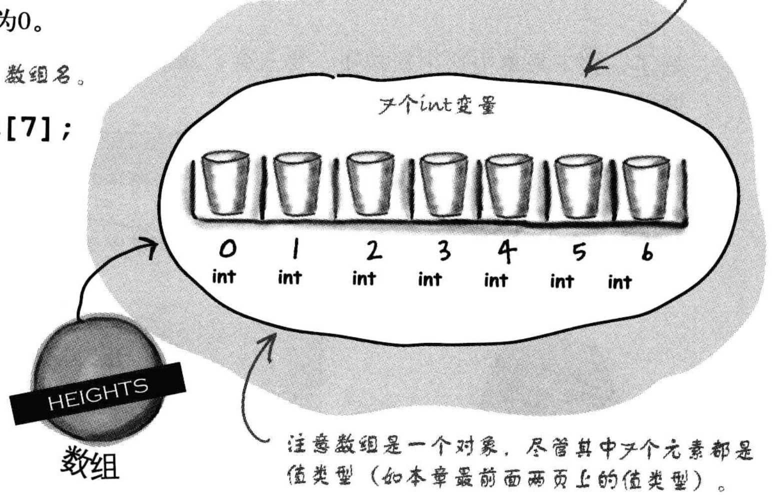
```
int[] heights;
heights = new int[7];
heights[0] = 68;
heights[1] = 70;
heights[2] = 63;
heights[3] = 60;
heights[4] = 58;
heights[5] = 72;
heights[6] = 74;
```

数组中各元素的类型。

数组名。

按索引来引用这些元素，不过每个元素实际上就像是一个正常的int变量。

在内存中，数组会作为一个内存块存储，尽管其中有多个int变量。




注意数组是一个对象，尽管其中7个元素都是值类型（如本章最前面两页上的值类型）。

# 数组还可以包含一组引用变量。

可以创建对象引用的一个数组，就像创建数值数组或串数组一样。数组并不关心其中存储的变量是何种类型，这由你来决定。所以可以有一个int数组，也可以有一个Duck对象数组，这没有任何问题。

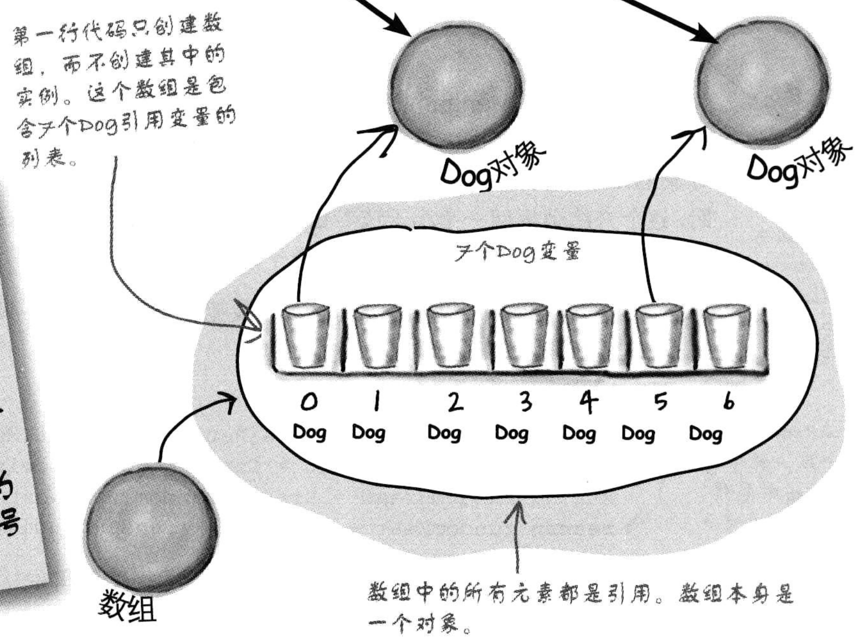
以下代码会创建一个包含7个Dog变量的数组。初始化数组的代码行只会创建引用变量。因为只有两行new Dog()代码，所以实际上只创建了Dog类的两个真正的实例。

**设置或获取数组中的元素时，中括号内的数字称为索引(index)。数组中第一个元素的索引为0。**

```
Dog[] dogs = new Dog[7];
dogs[5] = new Dog();
dogs[0] = new Dog();
```

这行代码声明一个dogs变量，用来保存Dog对象的引用数组，然后创建一个包含7个元素的数组。

这两行代码创建Dog()的新实例，并放在索引0和索引5的位置上。




## 数组的长度

可以使用数组的Length属性查出数组中有多少个元素。所以，如果有一个名为heights的数组，可以使用heights.Length得出这个数组的长度。如果数组中有7个元素，则得到长度为7，这说明数组元素的编号为从0到6。

# 欢迎品尝马虎Joe餐厅物美价廉的三明治！

马虎Joe预备了各式各样的肉，大量面包，另外调味料之多也超乎你的想象。可惜的是，他却没有一个菜单！你能构建一个程序，每天随机为他生成一个新菜单吗？

**动手做！**

| MenuMaker |
|---|
| Randomizer<br>Meats<br>Condiments<br>Breads |
| GetMenuItem() |

**1** **创建一个新工程，并增加一个MenuMaker类。**

要生成一个菜单，需要有配料，而用数组建立这种列表再合适不过了。我们还需要一种方法来随机地选择配料，把它们组合在一起构成一个三明治。幸运的是，.NET Framework提供了一个内置类，名为Random，它能生成随机数。这样一来，类中将有4个字段：Randomizer字段包含一个Random对象的引用，另外3个string数组分别存储肉、调味料和面包。

```
class MenuMaker {
    public Random Randomizer;

    string[] Meats = { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    string[] Condiments = { "yellow mustard", "brown mustard",
             "honey mustard", "mayo", "relish", "french dressing" };
    string[] Breads = { "rye", "white", "wheat", "pumpernickel",
             "italian bread", "a roll" };
}
```

Randomizer字段存储Random对象的一个引用。调用其Next()方法将生成随机数。

这个类有3个字段来存储3个不同的串数组。将用这些数组建立随机的菜单项。

记住，要使用中括号访问数组中的某个成员。Breads[2]的值是“wheat”。

**2** **为类增加一个GetMenItem()方法生成一个随机的三明治。**

注意到如何初始化这些数组了吗？这称为集合初始化方法（collection initializer），有关内容将在第8章详细介绍。

这个类的重点是生成三明治，所以下面增加一个方法来完成这个工作。它将使用Random对象的Next()方法从各个数组中随机选择肉、调味料和面包。向Next()方法传入一个int参数时，这个方法会返回一个小于该参数的随机数。所以，如果Random对象名为Randomizer，调用Randomizer.Next(7)将返回一个介于0到6之间的随机数。

那么怎么知道向Next()方法传入什么参数呢？这很容易——只需传入各数组的长度（Length）。这样就会返回数组中一个随机项的索引。

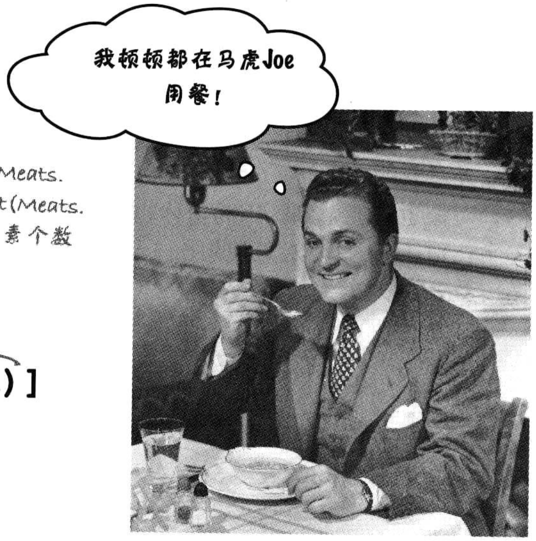
```
public string GetMenuItem() {
    string randomMeat = Meats[Randomizer.Next(Meats.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    return randomMeat + " with " + randomCondiment + " on " + randomBread;
}
```

GetMenuItem()方法返回一个串，其中包含由3个数组中随机选择的元素制成的一个三明治。

通过将Meats.Length传入Random对象的Next()方法，GetMenuItem()方法将取Meats数组中的一个随机项放入randomMeat。由于Meats数组中有5项，所以Meats.Length为5，因此Next(5)要返回介于0到4之间的一个随机数。

## 怎么做的……

randomizer.Next(7)方法会得到一个小于7的随机数。Meats.Length返回Meats中的元素个数。所以randomizer.Next(Meats.Length)会返回一个大于等于0但是小于Meats数组中元素个数的随机数。




```
Meats[Randomizer.Next(Meats.Length)]
```

Meats是一个串数组。其中有5个元素，编号为0到4。所以Meats[0]等于“Roast Beef”，Meats[3]等于“Ham”。

**3** **建立窗体。**

向窗体增加6个标签，label1到lable6。然后增加相应的代码，使用一个MenuMaker对象设置各个标签的Text属性。这里需要使用Random类的一个新实例来初始化这个对象。代码如下：

使用一个对象初始化方法，将MenuMaker对象的Randomizer字段设置为Random类的一个新实例。

```
public Form1() {
    InitializeComponent();

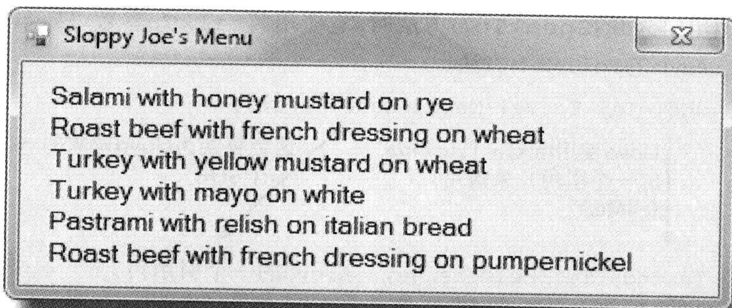
    MenuMaker menu = new MenuMaker() { Randomizer = new Random() };

    label1.Text = menu.GetMenuItem();
    label2.Text = menu.GetMenuItem();
    label3.Text = menu.GetMenuItem();
    label4.Text = menu.GetMenuItem();
    label5.Text = menu.GetMenuItem();
    label6.Text = menu.GetMenuItem();
}
```

现在使用GetMenuItem()方法随机生成6种不同的三明治。

这里需要再考虑一下。如果忘记初始化MenuMaker对象的Randomizer字段会怎么样呢？你能想出一种办法来避免这种情况发生吗？

运行这个程序时，6个标签将随机显示6种不同的三明治。



# 对象使用引用相互交谈

| Elephant |
|---|
| Name<br>EarSize |
| WhoAmI()<br>TellMe()<br>SpeakTo() |

到目前为止，我们已经看到，窗体使用引用变量与对象交互（调用对象的方法以及检查对象的字段）。对象还可以使用引用来相互调用方法。实际上，窗体能做的，对象也都能做到，因为窗体也是一个对象。对象交谈时，有一个很有用的关键字：this。只要对象使用this关键字，就是指它自己，这是一个引用，指向调用this的对象。

**1 以下方法让一个大象说话。**

下面向Elephant类增加一个方法。它的第一个参数是来自大象的一个消息，第二个参数是说这句话的大象：

```
public void TellMe(string message, Elephant whoSaidIt) {
    MessageBox.Show(whoSaidIt.Name + " says: " + message,);
}
```

调用这个方法时的代码如下。可以增加到button4 _ Click()，但是要把它增加到重置引用的语句(lloyd = lucinda;)前面！

```
lloyd.TellMe("Hi", lucinda);
```

调用Lloyd的TellMe()方法，并传入两个参数："Hi"和指向Lucinda对象的一个引用。这个方法使用whoSaidIt参数来访问作为第二个参数传入TellMe()的大象的Name字段。

**2 以下方法调用另一个方法。**

现在向Elephant类增加以下SpeakTo()方法。它使用了一个特殊的关键字：this。这是一个引用，允许对象自言自语。

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.TellMe(message, this);
}
```

Elephant类中的这个方法调用另一个大象的TalkTo()方法。它允许一个大象与另一个大象交谈。

下面仔细地分析这是如何做到的。

```
lloyd.SpeakTo(lucinda, "Hello");
```

调用Lloyd的SpeakTo()方法时，它使用其talkTo参数（Lucinda的一个引用）来调用Lucinda的TellMe()方法。

```
whoToTalkTo.TellMe(message, this);
```

Lloyd使用talkTo（Lucinda的一个引用）来调用TellMe()

这会替换为Lloyd对象的一个引用

```
lucinda.TellMe(message, [Lloyd的一个引用]);
```




所以就好像对Lucinda调用了TellMe("Hello", lloyd),，这会显示这样一个消息：

# 还没有对象

处理对象时还有一个重要的关键字会经常使用。创建一个新引用但未做任何设置时，它也有一个值。刚开始时会设置为null，这说明它不指向任何对象。

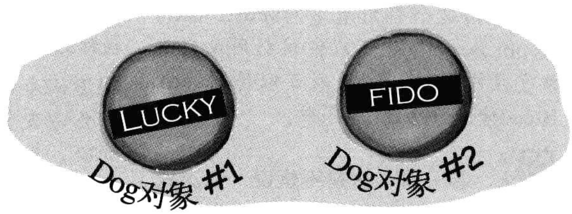
```
Dog fido;
Dog lucky = new Dog();
```

*目前，只有一个对象。fido引用设置为null。*



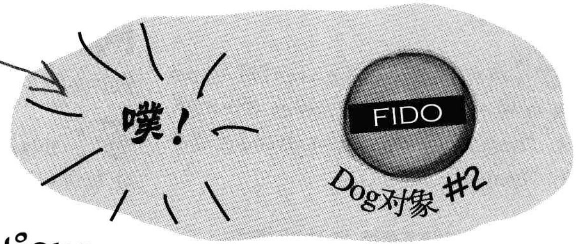

```
fido = new Dog();
```

*现在fido指向了一个对象，它不再等于null。*




```
lucky = null;
```

*设置lucky为null时，它不再指向原来的对象，所以会被垃圾回收。*




## there are no Dumb Questions

**问：** **再确认一次，窗体真的是一个对象吗？**

**答：** 没错！正因如此，你的类代码会以一个类声明开头。可以打开一个窗体的代码自己看看。然后打开你写的任何一个程序的Program.cs，查看Main()方法，你会发现“new Form1()”。

**问：** **为什么要使用null？**

**答：** 一般的程序中有几种情况会用到null，其中最常见的是测试：

```
if (lloyd == null) {
```

如果lloyd引用设置为null，则这个测试将返回true。

还有一种情况下会用到null关键字，如果希望将对象垃圾回收，就可以使用这个关键字。如果你有对象的一个引用，而且这个对象已经用完，将引用设置为null就会立即将对象标志为可以回收（除非这个对象在别处还有一个引用）。

**问：** **你一直在说垃圾回收，那么到底是谁在完成回收呢？**

**答：** 还记得我们在第1章开头谈到的通用语言运行时库（Common Language Runtime，CLR））吗？这是运行所有.NET程序的虚拟机。虚拟机（virtual machine）可以将运行程序与操作系统的其余部分相隔离。虚拟机的一个工作就是管理程序使用的内存。这说明，它会跟踪所有对象，明确对象的最后一个引用何时消失，然后释放该对象所用的内存。

## there are no Dumb Questions

**问：** **我还是不太确定引用是怎么工作的。**

**答：** 引用是使用对象方法和字段的唯一途径。如果创建了一个Dog对象的引用，就可以使用这个引用来访问为Dog对象创建的所有方法。如果有一个（非静态）Dog.Bark()或Dog.Beg()方法，可以创建一个名为spot的引用。然后使用这个引用来访问spot.Bark()或spot.Beg()。还可以使用引用改变对象字段中的信息。所以可以使用spot.Breed修改一个Breed字段。

**问：** **等一等，这是不是说，每次我通过一个引用改变一个值时，也在同时改变这个对象所有其他引用的这个值，是吗？**

**答：** 确实如此。如果rover引用与spot指向同一个对象，将rover.Breed改为"beagle"，那么spot.Breed也将等于"beagle"。

**问：** **我还是不太清楚不同类型存储不同大小的值，这到底是怎么回事？**

**答：** 是这样。关于变量，不论它的值有多大，都会按类型指定大小。所以，如果声明一个变量，并指定类型为long，尽管这个数非常小（比如说，只是一个5），CLR也会为它预留足够大的空间，以防它会变得很大。考虑到这一点会很有用。毕竟，变量之所以称为变量，就是因为它们总在改变。

CLR认为你知道自己在做什么，而且不会为变量指定一个不必要的类型。所以，即使这个数现在可能还不大，但是经过一些算术运算，它很可能改变，而CLR会留出足够大的内存来处理这种类型。

**问：** **我又想起来一个问题，"this"做什么用？**

**答：** this是一个特殊的变量，只能在对象内部使用。在一个类中，可以使用this指示这个特定实例中的任何字段或方法。处理一个类时，如果它的方法调用了另外一些类，this尤其有用。一个对象可以用this向另一个对象发送它自己的一个引用。所以，如果Spot调用了Rover的某个方法并传入this作为参数，就为Rover提供了Spot对象的一个引用。

**在对象实例化代码中，实例都可以使用这个特殊的this变量（其中包含它自己的一个引用）。**

## BULLET POINTS

确实有一种非常特殊的情况不需要声明类型，在第14章将学习何时使用"var"关键字。

- 声明一个变量时，总是要指定一个类型。有时可以将声明与赋值结合在一起。
- 对于数值变量有一些值类型（value types），可以存储不同大小的数。最大的数应当使用类型long，最小的数（不超过255）应当声明为byte。
- 每个值类型都有大小，不论数据的实际大小是多少，都不能把一个较大类型的值放在一个较小类型的变量中。
- 使用字面量值时，要使用F后缀指示float (15.6F)，使用M表示decimal (36.12M)。
- 有些类型C#知道如何自动转换（如short转换为int）。编译器不允许将一个变量设置为另一种类型的值时，就需要完成强制类型转换。
- 有一些词是C#的保留字，不能用这些保留字命名变量。这些保留字包括for、while、using、new以及C#语言中有特定作用的其他词。
- 引用就像标签，一个对象可以有多个引用，它们都指向同一个对象。
- 如果一个对象没有引用指向它，就会被垃圾回收。

## Sharpen your pencil

以下是Elephant对象的一个数组，并通过一个循环迭代处理这个数组，找出耳朵最大的大象。for循环每次迭代后biggestEars.Ears的值是什么？

```
private void button1_Click(object sender, EventArgs e)
{
  Elephant[] elephants = new Elephant[7];
  elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
  elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
  elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
  elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
  elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
  elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
  elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

  Elephant biggestEars = elephants[0];
  for (int i = 1; i < elephants.Length; i++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

创建一个包含7个Elephant()引用的数组。

每个数组都从索引0开始，所以数组中第一个大象是Elephants[0]。

这一行使biggestEars引用指向elephants[i]所指的大象。

当心，这个循环从数组的第二个元素（索引为1）开始，并迭代6次，直到i等于数组的长度。

Iteration #1 biggestEars.EarSize = ________

Iteration #2 biggestEars.EarSize = ________

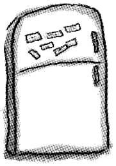
Iteration #3 biggestEars.EarSize = ________

Iteration #4 biggestEars.EarSize = ________

Iteration #5 biggestEars.EarSize = ________

Iteration #6 biggestEars.EarSize = ________

答案见166页。



# 代码磁贴

冰箱上胡乱贴了一个按钮的代码。你能重排这些代码片段，得到一个能正常工作并显示以下输出的方法吗？

```
int y = 0;
```

```
refNum = index[y];
```

```
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
```

```
int refNum;
while (y < 4) {
```

```
result += islands[refNum];
```

```
MessageBox.Show(result);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
}
```

```
}
```

```
string[] islands = new string[4];
```

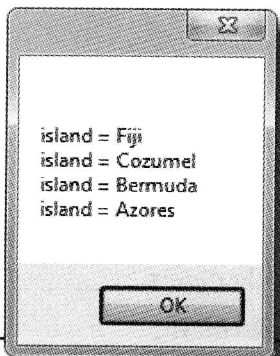
```
result += "\nisland = ";
```

```
int[] index = new int[4];
```

```
y = y + 1;
```

```
private void button1_Click (object sender, EventArgs e)
{
```

```
string result = "";
```



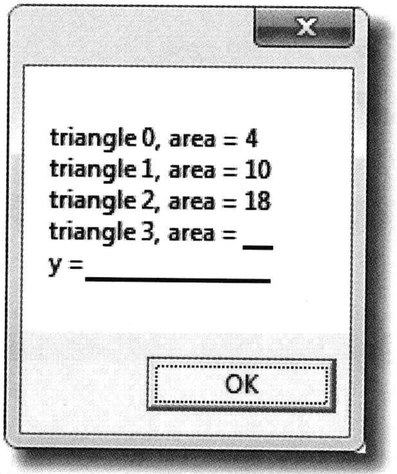

答案见167页。

# 池塘谜题



你的任务是从这个池塘里取出一些代码片段，在代码中填空。一个代码片段可以使用多次，另外不是所有片段都要用到。你的目标是建立一个能够编译并运行的类，要生成以下所列的输出。

**输出**




**附加问题！**

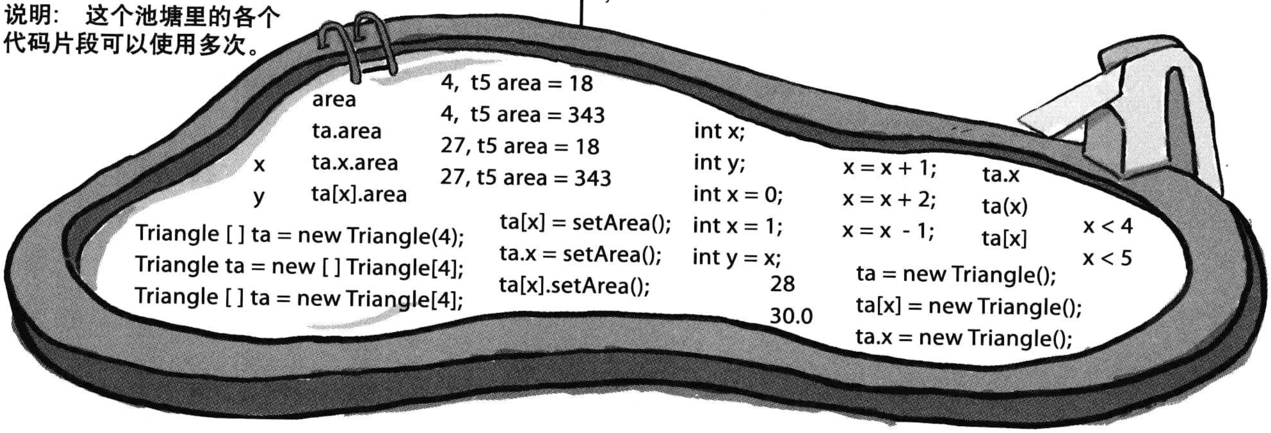
要想得到附加分，请使用池塘里的代码片段填上输出中的两个空。

```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";
        __________
        ____________________________
        while ( ________ )
        {
            __________________________
            _____.height = (x + 1) * 2;
            _____.length = x + 4;
            __________________
            results += "triangle " + x + ", area";
            results += " = " + _____.area + "\n";
            ___________
        }
        ___________
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        ____________ = (height * length) / 2;
    }
}
```

这是应用的入口点。假设它所在的文件最前面有正确的"using"代码行。

**提示：** SetArea()不是一个静态(static)方法。翻到第3章回忆一下static关键字是什么意思。

**说明：** 这个池塘里的各个代码片段可以使用多次。




答案见168页。

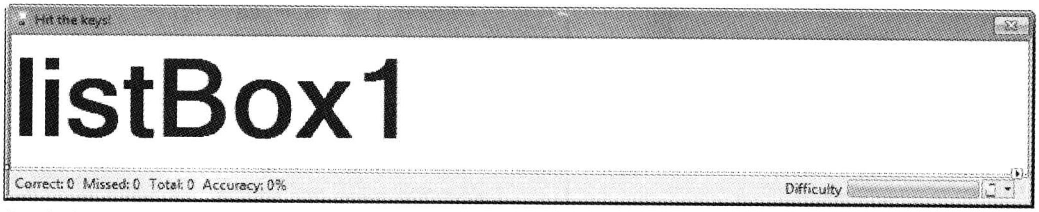
# 构建一个打字游戏

你已经达到了一个里程碑……到目前为止，你已经掌握了足够的知道，可以构建一个游戏了！游戏的工作如下。窗体会显示一些随机的字母。如果玩家正确地输入了其中某个字母，这个字母就会消失，正确率会增加。如果玩家输入了错误的字母，正确率会降低。随着玩家输入更多的字母，游戏会变得越来越快，使是输入各个正确字母的难度越来越大。如果窗体已经填满了字母，则游戏结束！

**动手做！**




1. **建立窗体。**
在窗体设计工具中，要创建的窗体如下：



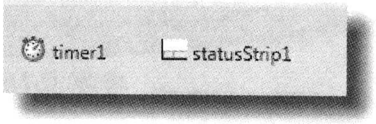

你需要：

★ 去掉最小框和最大框。然后设置窗体的**FormBorderStyle**属性为**Fixed3D**。这样一来，玩家就不会无意地拖动窗体或者调整窗体的大小了。再将窗体大小调整为宽度远远大于高度（我们将窗体的大小设置为宽度为876，高度为174）。

★ 将一个列表框（**ListBox**）从工具箱拖到窗体上。将这个列表框的**Dock**属性设置为Fill，**MultiColumn**属性设置为True。再将**Font**设置为72点大小粗体。

★ 在工具箱中，展开上面的“All Windows Forms”组。这会显示很多控件。找到定时器（Timer）控件，双击这个控件把它增加到窗体上。

★ 在工具箱的“All Windows Forms”组中找到**StatusStrip**，双击这个控件，为窗体增加一个状态条。现在应该能看到窗体设计工具下方灰色区域中的**StatusStrip**和**Timer**图标：




**看到了吗？这里使用一个Timer可以让窗体同时做多件事情。花点时间翻到附录“其他”中的#3，了解还可以采用另一种方法做到这一点。**

**Relax**

**你将使用3个新控件，不过这些控件都很容易使用！**

即使你之前从未见过列表框控件（ListBox）、状态条控件（StatusStrip）或定时器控件（Timer），起码已经知道如何设置它们的属性以及在代码中如何使用。在接下来的几章将学习更多有关内容。

2 **设置StatusStrip控件。**

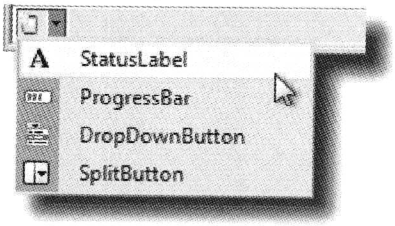
再仔细看看截屏图下方的状态条。这个状态条的一边有一系列标签：




另一边则有一个标签和一个进度条：




单击StatusStrip的下拉列表选择StatusLabel来增加一个StatusLabel：




- 将`StatusStrip`的`SizingGrip`属性设置为False。
- 使用Properties窗口将其（Name）设置为`correctLabel`，并设置`Text`为“Correct: 0”。再增加3个`StatusLabels`：`missedLabel`、`totalLabel`和`accuracyLabel`。
- 再增加一个`StatusLabel`。将其`Spring`设置为`True`，`TextAlign`设置为`MiddleRight`，另外将`Text`设置为“Difficulty”。最后，增加一个`ProgressBar`，命名为`difficultyProgressBar`。

3 **设置Timer控件。**

注意到了吗？Timer控件并没有出现在窗体上。这是因为Timer是一个不可见的控件。它不会具体改变窗体的外观。实际上它只做一件事：反复调用一个方法。将Timer控件的**`Interval`**属性设置为800，这样它会每800毫秒调用一次方法。然后在窗体设计工具中双击timer1图标。IDE会像平常一样，当你双击某个控件时它总会向窗体增加一个方法。这一次，它会增加一个名为**`timer1 _ Tick`**的方法。这个方法的代码如下：

```
private void timer1_Tick(object sender, EventArgs e)
{
    // Add a random key to the ListBox
    listBox1.Items.Add((Keys)random.Next(65, 90));
    if (listBox1.Items.Count > 7)
    {
        listBox1.Items.Clear();
        listBox1.Items.Add("Game over");
        timer1.Stop();
    }
}
```

稍后将增加一个名为“random”的字段。你能猜出它的类型吗？

**4** **增加一个类来跟踪玩家的统计信息。**

如果窗体要显示玩家的按键总数、错误和正确的按键数，以及玩家的正确率，我们就需要一种方法来跟踪所有这些数据。听上去需要建立一个新类来完成这个工作！下面向工程中增加一个名为Stats的类。这个类有4个int字段，分别为：Total、Missed、Correct和Accuracy，另外还包括一个名为Update的方法，它有一个bool参数：如果玩家输入ListBox中有的一个正确字母，则这个方法返回true，如果玩家输入有误，则返回false。

| Stats |
|---|
| Total<br>Missed<br>Correct<br>Accuracy |
| Update() |

```
class Stats
{
    public int Total = 0;
    public int Missed = 0;
    public int Correct = 0;
    public int Accuracy = 0;

    public void Update(bool correctKey)
    {
        Total++;

        if (!correctKey)
        {
            Missed++;
        }
        else
        {
            Correct++;
        }

        Accuracy = 100 * Correct / (Missed + Correct);
    }
}
```

*每次调用Update()方法时，都会重新计算正确率，并放在Accuracy字段中。*

**5** **向窗体中增加字段来存放一个Stats对象和一个Random对象。**

需要这个新**Stats**类的一个实例来具体存储信息，因此增加一个名为**stats**的字段存储有关信息。前面已经看到，我们需要一个名为**random**的字段，其中将包含一个**Random**对象。

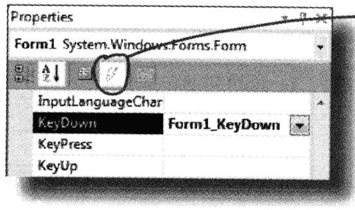
在窗体最前面增加这两个字段：

```
public partial class Form1 : Form
{
    Random random = new Random();
    Stats stats = new Stats();
    ...
```

## 6 处理按键。

这个游戏还有最后一件事要做：只要玩家按下一个键，就需要检查这个按键是否正确（如果正确则从ListBox中将这个字母删除），并更新StatusStrip上的统计信息。

回到窗体设计工具，选择这个窗体。然后在Properties窗口中单击闪电按钮。滚动到KeyDown行，双击这一行。这会告诉IDE增加名为Form1 _ KeyDown()的方法，每次用户按下一个键时都将调用这个方法。这个方法的代码如下：




单击这个按钮，改变Properties窗口的视图。它左边的按钮可以将Properties窗口切换回原来的视图，显示控件的属性。

这些称为事件（event），后面将学习有关事件的更多内容。

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    // If the user pressed a key that's in the ListBox, remove it
    // and then make the game a little faster
    if (listBox1.Items.Contains(e.KeyCode))
    {
        listBox1.Items.Remove(e.KeyCode);
        listBox1.Refresh();
        if (timer1.Interval > 400)
            timer1.Interval -= 10;
        if (timer1.Interval > 250)
            timer1.Interval -= 7;
        if (timer1.Interval > 100)
            timer1.Interval -= 2;
        difficultyProgressBar.Value = 800 - timer1.Interval;

        // The user pressed a correct key, so update the Stats object
        // by calling its Update() method with the argument true
        stats.Update(true);
    }
    else
    {
        // The user pressed an incorrect key, so update the Stats object
        // by calling its Update() method with the argument false
        stats.Update(false);
    }

    // Update the labels on the StatusStrip
    correctLabel.Text = "Correct: " + stats.Correct;
    missedLabel.Text = "Missed: " + stats.Missed;
    totalLabel.Text = "Total: " + stats.Total;
    accuracyLabel.Text = "Accuracy: " + stats.Accuracy + "%";
}
```

这个if语句检查ListBox来查看其中是否包含玩家按下的按键。如果确实包含，则将这个字母从ListBox中删除，并增加游戏的难度。

随着玩家输入更多的正确字母，这一部分会增加游戏的难度。你可以减少从timer1.Interval减去的量，让游戏更容易一些，或者也可以增加这个量让游戏难度更大。

玩家按一个按键时，Form1_KeyDown()方法会调用Stats对象的Update()方法，来更新玩家的统计信息，然后显示在StatusStrip中。

## 7 运行游戏。

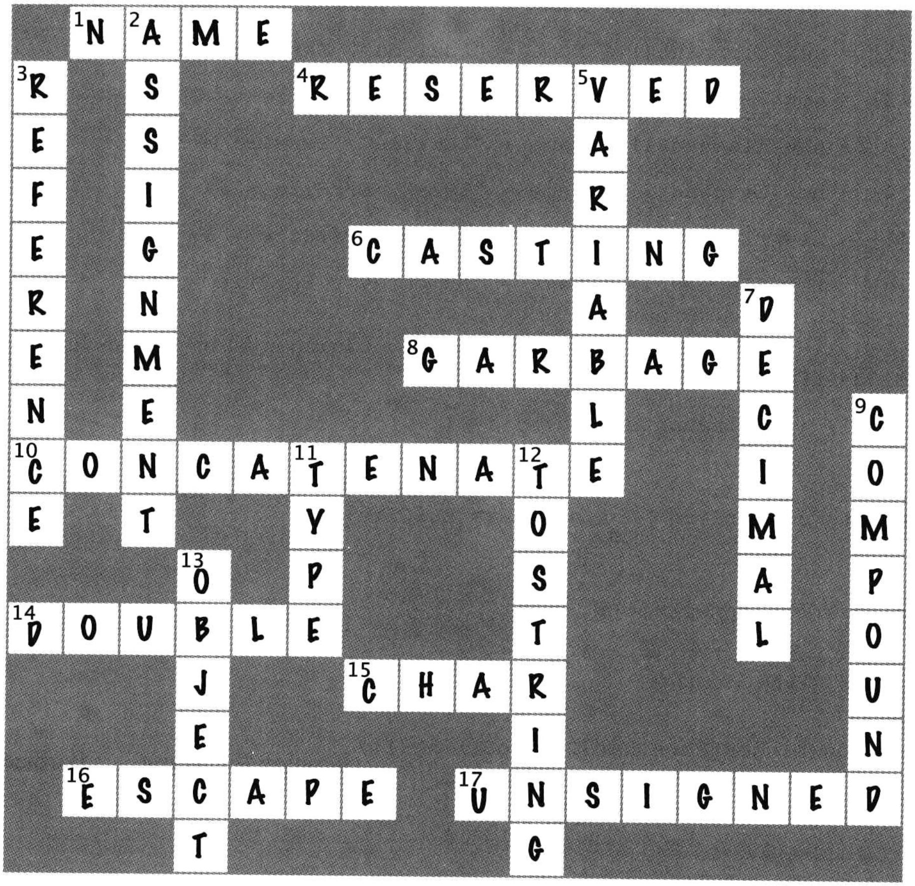
游戏完成了！试一试看看你做得怎么样。你可能需要调整ListBox的字体大小，确保其中刚好包含7个字母，还可以在Form1 _ KeyDown()方法中调整从timer1.Interval减去的值来改变游戏的难度。

## Exercise Solution

C#中有大约77个保留字。它们由C#编译器所保留，不能用做变量名。读完这本书后你就会了解所有这些保留字。以下给出前面已经用到的一些保留字。你认为这些保留字在C#中起什么作用？请写下来。

| | |
|---|---|
| **namespace** | namespace确保你在程序中使用的名字不会与.NET Framework或程序中使用的其他外部类中的名字相冲突。一个程序中的所有类和方法都在一个命名空间中。 |
| **for** | for允许完成一个循环，其中执行3个语句。首先声明要使用的变量，后面的语句用这个变量来计算一个条件是否成立。第三个语句对这个值做某种处理。 |
| **class** | 要利用class来定义对象。类包含属性和方法。属性是对象知道的东西，而方法是对象完成的事情。 |
| **public** | 公共类可以由工程中所有其他类使用，如果一个变量或方法声明为公共（public），则除了声明该变量的方法外，还可以在其他类中使用或由其他方法调用。 |
| **else** | 以else开头的代码会在它前面的if语句失败时执行。 |
| **new** | 使用这个关键字来创建对象的一个新实例。 |
| **using** | 利用using，可以列出程序中使用的所有命名空间。这样一来，除了你自己创建的类外，还允许使用.NET Framework以及第三方预定义类的代码。 |
| **if** | 这是在程序中建立条件语句的一种方法。表示如果一个条件为true，则做某件事，否则做另外一件事。 |
| **while** | while循环是一种“当”循环：当其中的条件为true时将持续执行。 |

# 类型填字游戏答案


1 NAME
2 ASSIGNMENT
3 REFERENCE
4 RESERVED
5 VARIABLE
6 CASTING
7 DECIMAL
8 GARBAGE
9 COMPOUND
10 CONCATENATE
11 TYPE
12 TOSTRING
13 OBJECT
14 DOUBLE
15 CHAR
16 ESCAPE
17 UNSIGNED

## Sharpen your pencil Solution

以下是Elephant对象的一个数组，并通过一个循环迭代处理这个数组，找出耳朵最大的大象。for循环每次迭代后biggestEars.Ears的值是什么？

```
private void button1_Click(object sender, EventArgs e)
{
  Elephant[] elephants = new Elephant[7];
  elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
  elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
  elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
  elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
  elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
  elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
  elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

  Elephant biggestEars = elephants[0];
  for (int i = 1; i < elephants.Length; i++)
  {
    if (elephants[i].EarSize > biggestEars.EarSize)
    {
      biggestEars = elephants[i];
    }
  }
  MessageBox.Show(biggestEars.EarSize.ToString());
}
```

还记得这个循环从数组第二个元素开始吗？为什么是这样？

biggestEars 引用用于跟踪目前为止for循环中找到的耳朵最大的元素（大象）。

使用调试工具来检查这一点！在这里设置断点查看biggestEars.EarSize。

for循环从第二个大象开始，将它与biggestEars指向的大象相比较。如果它的耳朵更大，则让biggestEars指向这个大象。然后再比较下一个大象，再下一个大象……到循环结束时，biggestEars就会指向耳朵最大的大象。

Iteration #1 biggestEars.EarSize = 40

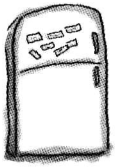
Iteration #2 biggestEars.EarSize = 42

Iteration #3 biggestEars.EarSize = 42

Iteration #4 biggestEars.EarSize = 44

Iteration #5 biggestEars.EarSize = 44

Iteration #6 biggestEars.EarSize = 45



# 代码磁贴

冰箱上胡乱贴了一个按钮的代码。你能重排这些代码片段，得到一个能正常工作并显示以下输出的方法吗？

```
private void button1_Click (object sender, EventArgs e)
{

string result = "";

int[] index = new int[4];

index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;

string[] islands = new string[4];

islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";

int y = 0;

int refNum;
while (y < 4) {

refNum = index[y];

result += "\nisland = ";

result += islands[refNum];

y = y + 1;

}

MessageBox.Show(result);

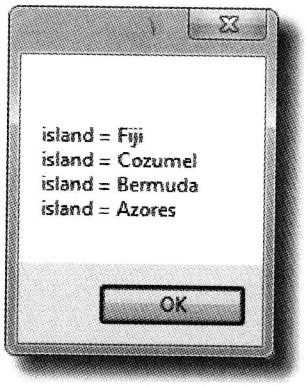
}
```

index[]数组在这里初始化。

islands[]数组在这里初始化。

通过使用 += 操作符完成连接来建立result串。

这个while循环从index[]数组取一个值，用做islands[]数组中的索引。



# 池塘谜题答案



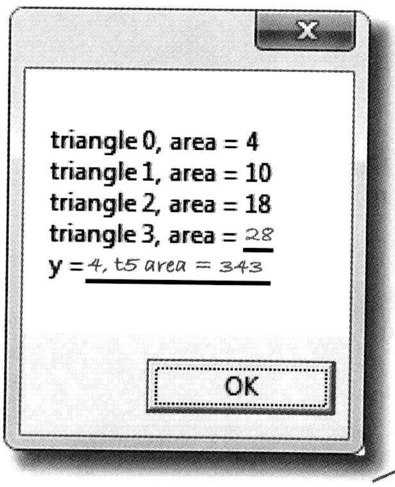
注意，这个类包含入口点，居然还创建了它自己的一个实例？这在C#中是完全合法的。

这一行之后，就得到了一个包含4个Triangle引用的数组，但是还没有任何Triangle对象！

while循环通过调用4次new语句，创建了4个Triangle实例。

```
class Triangle
{
    double area;
    int height;
    int length;
    public static void Main(string[] args)
    {
        string results = "";
        int x = 0;
        Triangle[] ta = new Triangle[4];
        while ( x < 4 )
        {
            ta[x] = new Triangle();
            ta[x].height = (x + 1) * 2;
            ta[x].length = x + 4;
            ta[x].setArea();
            results += "triangle " + x + ", area";
            results += " = " + ta[x].area + "\n";
            x = x + 1;
        }
        int y = x;
        x = 27;
        Triangle t5 = ta[2];
        ta[2].area = 343;
        results += "y = " + y;
        MessageBox.Show(results +
            ", t5 area = " + t5.area);
    }
    void setArea()
    {
        area = (height * length) / 2;
    }
}
```

## 附加题答案




SetArea()方法使用height和length字段来设置area字段。因为这不是一个静态方法，所以只能从Triangle的一个实例调用。

姓 名: 日 期:

# C#实验室

## 赛狗日

这个实验室将给出一个程序的规范，要求你根据前几章所学的知识构建这个程序。

这个工程比你之前见过的所有工程规模更大。所以在着手开发之前，先花点时间了解整个问题，磨刀不误砍柴工。另外，如果你被某个问题卡住了也不用担心，这里并没有新内容，所以完全可以继续向下读，以后再回来完成这个实验室。

我们已经为你完成了一些设计细节，而且可以保证已经提供了你需要的全部知识……再没有其他可以补充的了。

你要自己来完成这个任务。可以从网站下载这个实验室的一个可执行文件……不过我们不会提供这个工程的源代码。

# 规范：构建一个赛狗模拟系统

Joe、Bob和Al热衷于赛狗，不过他们可不想把钱都输光。需要你为他们构建一个模拟系统，以便在下注之前知道谁是赢家。另外，如果你干得漂亮，他们还会给你分红。

以下是要为他们建立的对象……

## 人

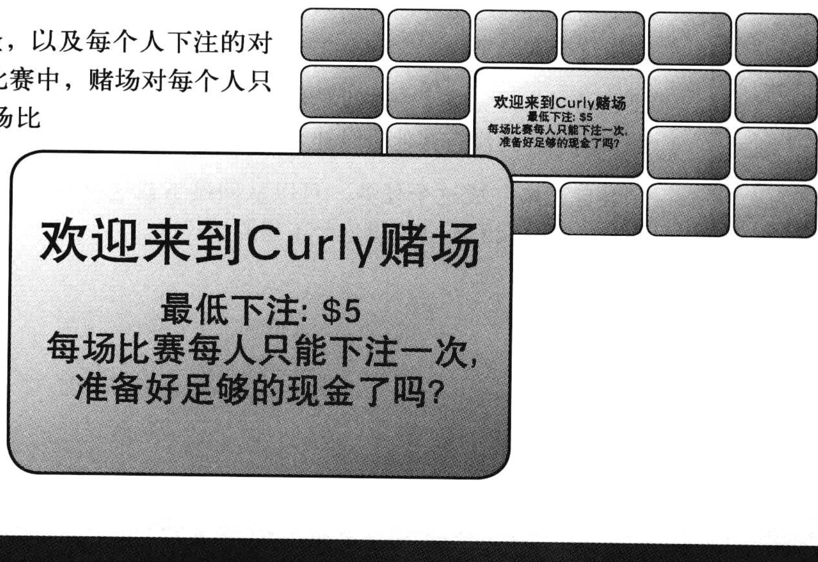
Joe、Bob和Al希望参加赛狗赌博。Joe开始有50元，Bob开始有75元，Al最初有45元。每次比赛前，他们都会各自决定是否下注以及所押的赌金。直到比赛前，他们都可以改变赌金…… 但是一旦比赛开始，赌金就再不能更改了。



## 赌场

赌场会跟踪每个人持有的现金，以及每个人下注的对象。每次下注至少5元。一场比赛中，赌场对每个人只取一次赌金，也就是说，每场比赛每个人不能重复下注。

赌场会检查下注的人确实有足够的现金付他的赌金，所以如果没有钱来做赌资，这个人就不能下注。



## 下注

每次下注都会“翻倍或全赔”，如果赢了，他的钱会翻倍，否则就会输掉这次的赌金。要求最低赌金为5元，而且每个人对一只狗最多下注15元。如果这只狗赢了，下注的人就会获得赌金的双倍（当然是在比赛结束之后）。如果他输了，就会将这次的赌金扣除。

**所有赌金：翻倍或全赔**
**最低赌金：$5**
**每只狗最多押$15**
**赢家：增加$**
**输家：扣除$**

假设有个人对window（一只狗）押了$10。比赛结束时，如果这只狗赢了，则他的现金就会增加$10（除了保留原来的$10赌金，还要加上赢来的另外$10）。如果他输了，则现金就会少$10。

## 比赛

有4只狗在直道上比赛。比赛胜者是第一只穿过终点线的狗。比赛完全是随机的，没有特设的不利条件或加分，一只狗以往的表现好并不意味着它获胜的机会更大。

如果你想构建一个指定了限制条件的系统，完全可以动手试试看！这是一个非常好的练习，可以编写一些很有意思的代码。



**听上去挺有意思？后面还有更多详细说明……**

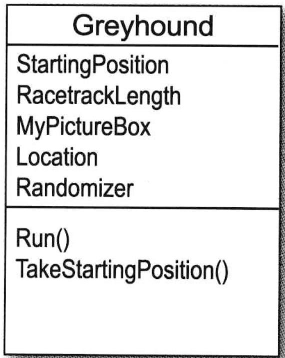
# 需要3个类和一个窗体

需要在Greyhound和Guy类最上面增加“using System.Windows.Forms”。另外还要为Greyhound增加“using System.Drawing;”，因为它会使用Point类。

这个工程里要构建3个主要的类，还要有这个模拟系统的一个GUI。应当有一个包含3个Guy对象的数组跟踪这3个人和他们的输赢情况，还要有一个数组包含4个真正参与比赛的Greyhound对象。另外，Guy的各个实例应当有自己的Bet对象，来跟踪他下注的情况，并在比赛结束后扣除（或增加）现金。

我们已经给出了类描述以及一些代码段，可以以此为起点。其他的所有代码都要由你来完成。

我们已经提供了所要构建的类的骨架。你的任务是具体填写这些方法。




```
class Greyhound {
  public int StartingPosition; // Where my PictureBox starts
  public int RacetrackLength; // How long the racetrack is
  public PictureBox MyPictureBox = null; // My PictureBox object
  public int Location = 0; // My Location on the racetrack
  public Random Randomizer; // An instance of Random

  public bool Run() {
    // Move forward either 1, 2, 3 or 4 spaces at random
    // Update the position of my PictureBox on the form
    // Return true if I won the race
  }

  public void TakeStartingPosition() {
    // Reset my location to the start line
  }
}
```

只需要一个Random实例，每个Greyhound的Randomizer引用都应当指向同一个Random对象。

这里增加了注释，以便你了解要做什么。

不要想得过于复杂……有时可能只需设置一个变量，仅此而已。

看看类图与代码如何对应？

Greyhound对象初始化方法很简单。只要确保为各个Greyhound对象传入窗体上相应PictureBox的一个引用。

## 你的对象可以控制窗体上的状况……

必须确保窗体将正确的图片框传入各个Greyhound对象初始化方法。

Greyhound类跟踪比赛中狗在赛道上的位置。它还会更新相应PictureBox的位置（表示狗沿着赛道向前跑）。Greyhound的各个实例使用一个名为MyPictureBox的字段来引用窗体上显示狗图片的PictureBox控件。假设distance变量包含狗前进的距离。以下代码将更新MyPictureBox的位置，让它的X值增加distance：

```
Point p = MyPictureBox.Location;
p.X += distance;
MyPictureBox.Location = p;
```

得到图片的当前位置……

……将其X坐标增加这个值，使它向前跑……

……然后更新窗体上图片框的位置。

**Guy**

Name
MyBet
Cash
MyRadioButton
MyLabel

UpdateLabels()
PlaceBet()
ClearBet()
Collect()

```
class Guy {
   public string Name; // The guy's name
   public Bet MyBet; // An instance of Bet() that has his bet
   public int Cash; // How much cash he has

   // The last two fields are the guy's GUI controls on the form
   public RadioButton MyRadioButton; // My RadioButton
   public Label MyLabel; // My Label

   public void UpdateLabels() {
     // Set my label to my bet's description, and the label on my
     // radio button to show my cash ("Joe has 43 bucks")
   }v

   public void ClearBet() { } // Reset my bet so it's zero

   public bool PlaceBet(int Amount, int Dog) {
     // Place a new bet and store it in my bet field
     // Return true if the guy had enough money to bet
   }

   public void Collect(int Winner) { } // Ask my bet to pay out
}
```

一旦设置MyLabel为窗体上某个标签，就可以使用MyLabel.Text改变这个标签的文本。MyRadioButton也是如此！

初始化Guy对象时，确保将其MyBet字段设置为**null**，一旦初始化则调用UpdateLabels()方法。

在这里增加你的代码。

记住，赌注由Bet的实例表示。

这里的关键是使用Bet对象……让它来完成工作。

这就是应用中Guy用来表示赌注的对象。

Bet的对象初始化方法只是设置amount、dog和bettor。

**Bet**

Amount
Dog
Bettor

GetDescription
PayOut

```
class Bet {
   public int Amount; // The amount of cash that was bet
   public int Dog; // The number of the dog the bet is on
   public Guy Bettor; // The guy who placed the bet

   public string GetDescription() {
     // Return a string that says who placed the bet, how much
     // cash was bet, and which dog he bet on ( "Joe bets 8 on
     // dog #4" ). If the amount is zero, no bet was placed
     // ( "Joe hasn' t placed a bet" ).
   }

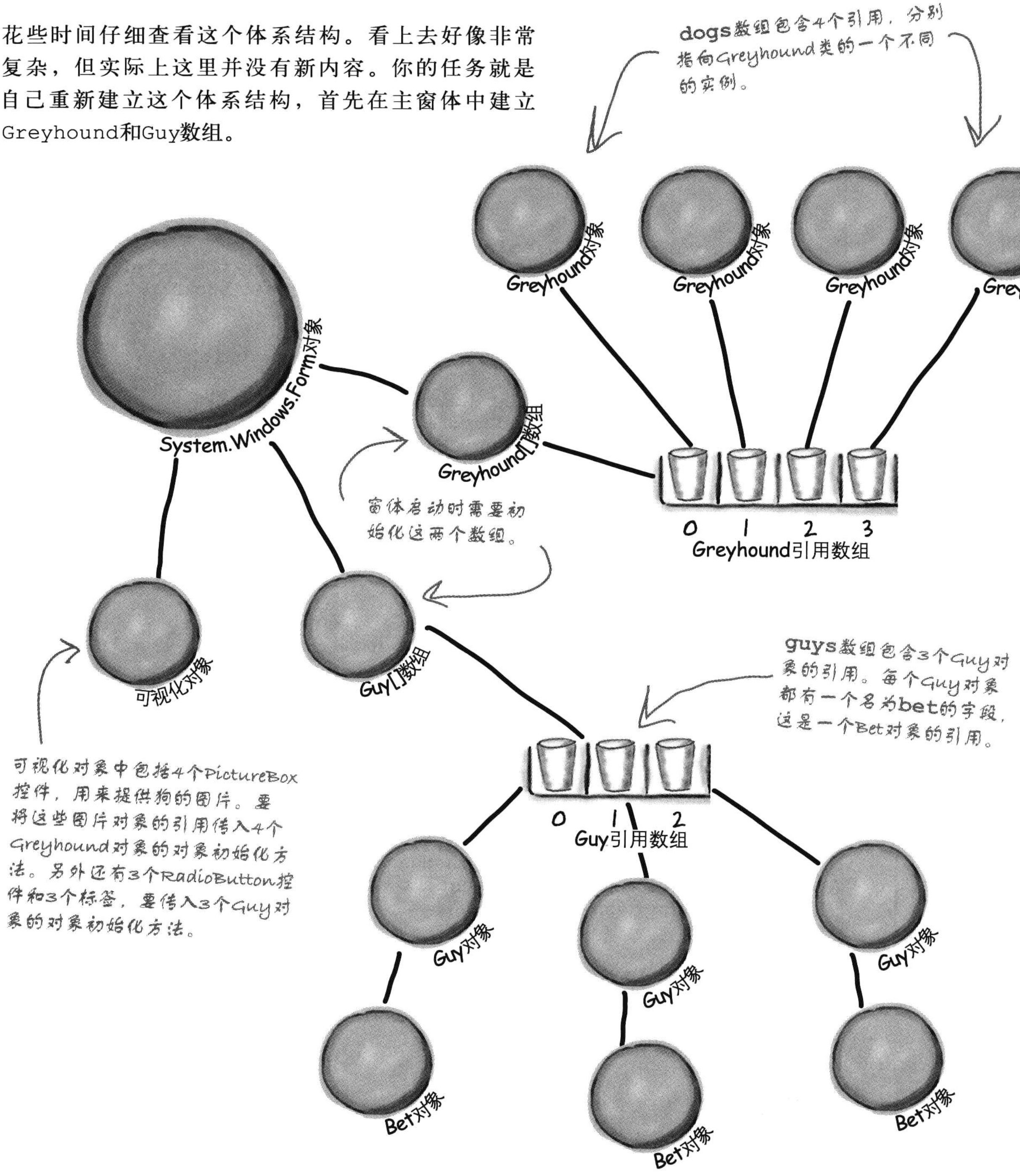
   public int PayOut(int Winner) {
     // The parameter is the winner of the race. If the dog won,
     // return the amount bet. Otherwise, return the negative of
     // the amount bet.
   }
```

这是一个常见的编程任务：根据多个数据合成一个串或消息。

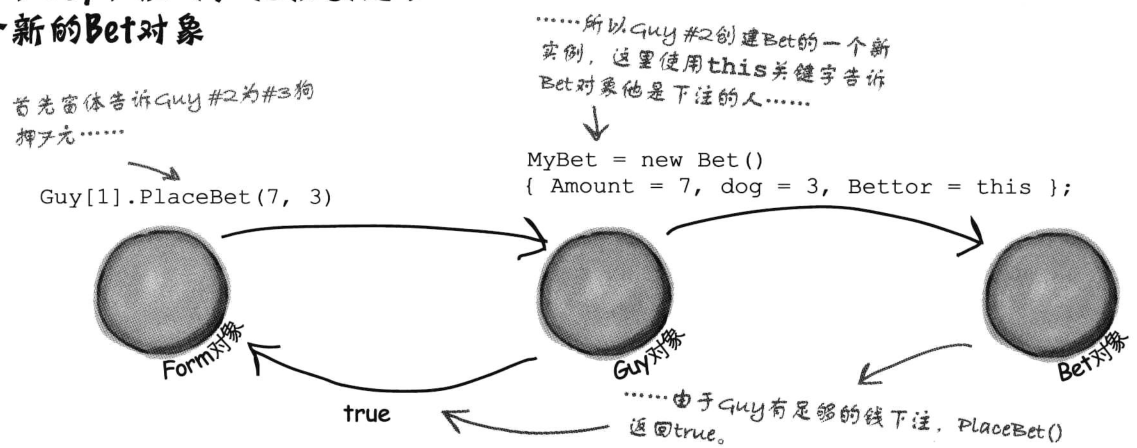
提示：你将在Guy代码中实例化Bet。Guy会使用**this**关键字向Bet的初始化方法传入Guy自己的一个引用。

# 以下是应用的体系结构

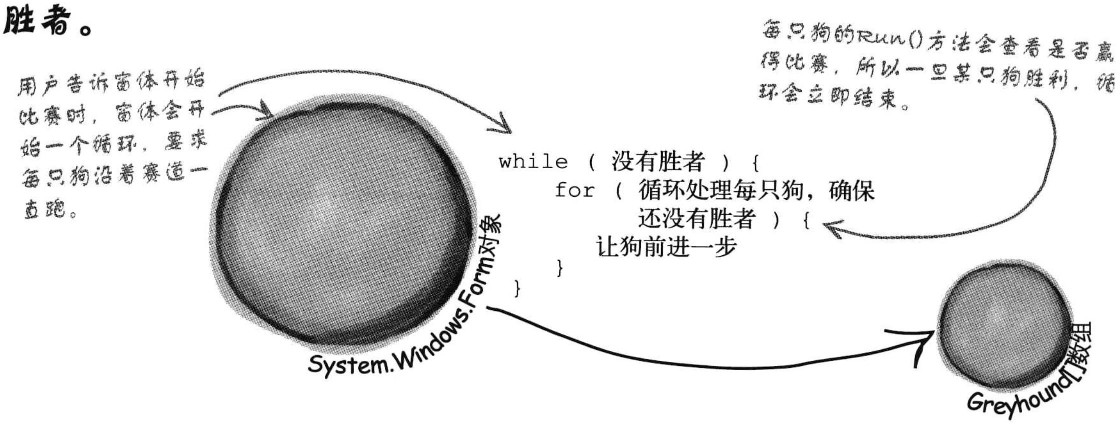
花些时间仔细查看这个体系结构。看上去好像非常复杂，但实际上这里并没有新内容。你的任务就是自己重新建立这个体系结构，首先在主窗体中建立Greyhound和Guy数组。



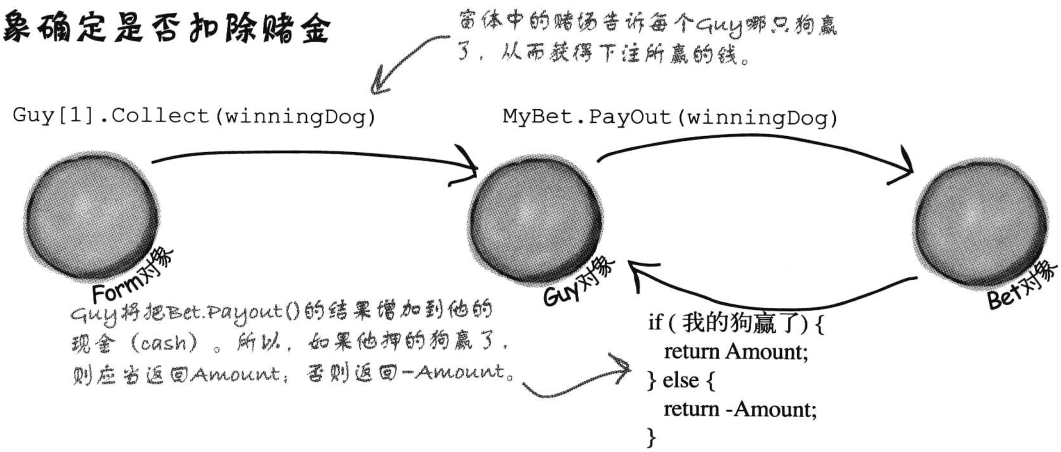
## 一个Guy下注时，他就创建了一个新的Bet对象




## 窗体告诉狗要一直跑，直到出现胜者。

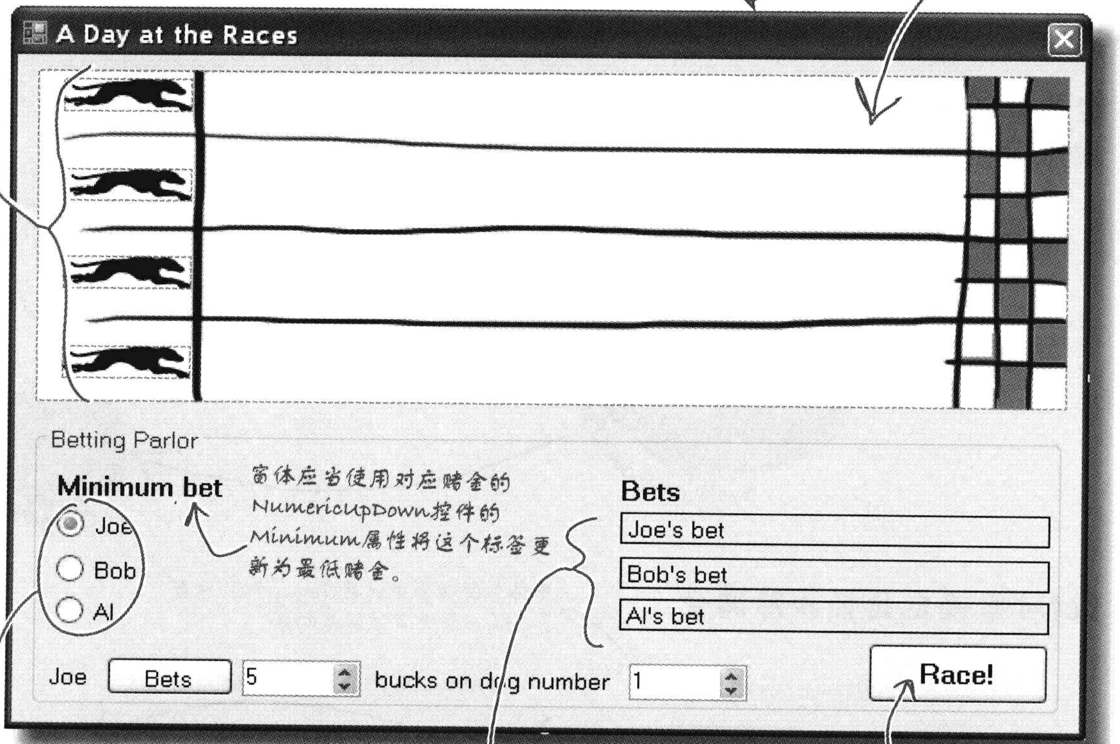



## Bet对象确定是否扣除赌金



# GUI的外观

“赛狗日”应用的图形用户界面（graphical user interface，GUI）包括一个窗体，其中划分为两部分。上面是赛道：对应赛道有一个PictureBox控件，对应4只狗还有另外4个PictureBox控件。窗体的下半部分显示了赌场，在这里3个人（Joe，Bob和Al）可以对比赛的结果下注。




**可以从www.headfirstlabs.com/books/hfcsharp/下载图片文件。**

# 下注

使用Betting Parlor（赌场）分组框中的控件完成每个人的下注。这里有3个阶段：

**1 尚未下注。**

程序第一次启动时，或者如果一场比赛刚刚结束，赌场里没有任何下注。可以看到每个人的现金总额（左边是相应人名）。

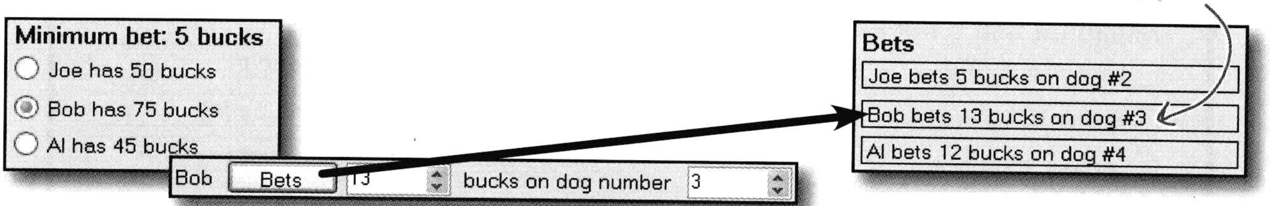
一个人下注时，他的Guy对象会使用MyLabel引用更新这个标签。还会使用MyRadioButton引用更新他的现金。




**2 每个人下注。**

要完成下注，选择这个人对应的单选钮，再选择一定金额和某只狗，然后单击Bets按钮。相应的PlaceBet()方法会更新标签和单选钮。

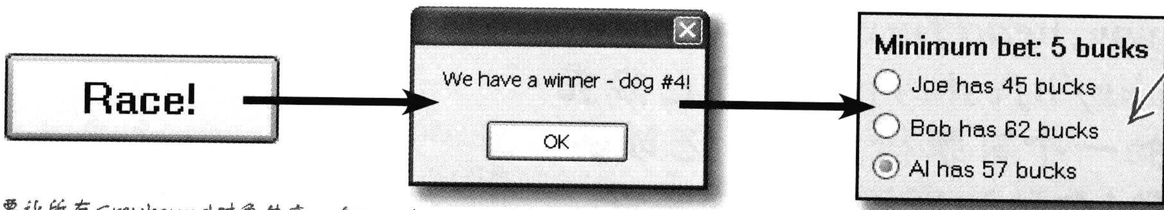
一旦Bob完成下注，他的Guy对象会更新这个标签和单选钮文本。




**3 比赛结束之后，每个人拿到他赢的钱（或扣除赌注）！**

一旦比赛结束，而且有一个胜者，每个Guy对象就会调用Collect()方法，完成现金增减。

由于Al为获胜的狗押了12元，他的现金会增加12元。另外两个人则会把赌金输掉。

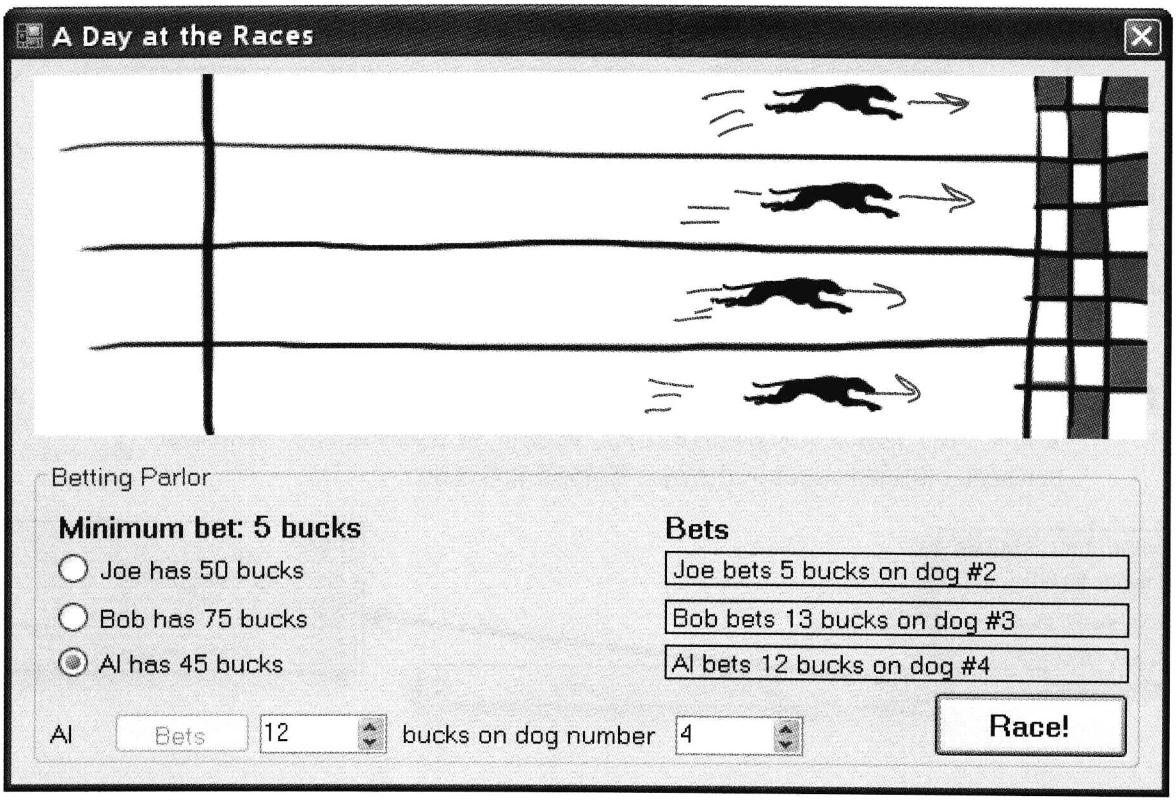



一定要让所有Greyhound对象共享一个Random！如果每只狗都创建自己的一个新Random实例，可能会看到一个bug：所有狗都生成同样的随机数序列。

# 最终产品

应该知道，如果每个人都能下注，并能观察赛狗现况，这个“赛狗日”（Day at the Races）应用就完成了。

比赛中，4个狗图片一直在赛道上奔跑，直到其中一只赢得比赛。




比赛过程中不允许下注……，要确保狗正在奔跑时不能开始一场新的比赛。

**可以从Head First labs网站（www.headfirstlabs.com/books/hfcsharp）下载已经完成的一个可执行文件，还可以下载4个狗和赛道的图片文件。**

不过你找不到源代码！在实际生活中，你不会得到编程问题现成的解决方案！这里提供了一个很好的机会，你可以借此检验你的C#知识掌握情况，看看已经学到了多少。

# 5 封装

# 让你的隐私……属于你个人



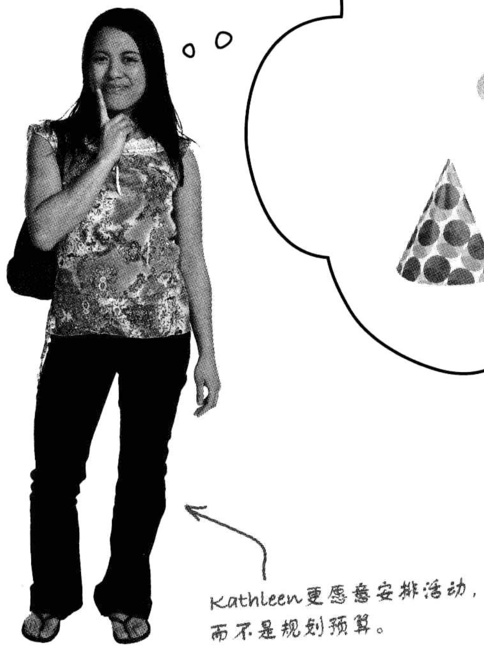

**是不是想要更多点隐私？**

有时你的对象也这么想。你肯定不希望不信任的人看你的杂志或者随意翻看你的银行票据。同你一样，好对象也不会让别的对象随意侵入它们的领地。这一章中，你将了解到封装的威力，你会把对象的数据设置为私有，并增加方法对这些数据的访问加以保护。

# Kathleen是一个策划人

她一直在为客户筹备宴会，而且干得不错。不过，她最近遇到了点麻烦，需要根据客户的要求尽快地做出预算。






一个新客户打电话请Kathleen准备一次宴会时，她需要知道有多少个客人参加，需要提供哪些种类的饮料，另外需要买哪些装饰品。然后根据一个流程图（多年来她一直在使用这个流程图），通过一个相当复杂的计算得出总花费。糟糕的是，按这个流程图完成计算花了她太长时间，当她还在做预算时，客户已经等不及开始考虑让其他策划人接手。

你要为她构建一个C#驱动的活动预算工具，来保住这笔业务。如果你办到了，则想想看她一定会为你举办一次超级盛大的宴会。

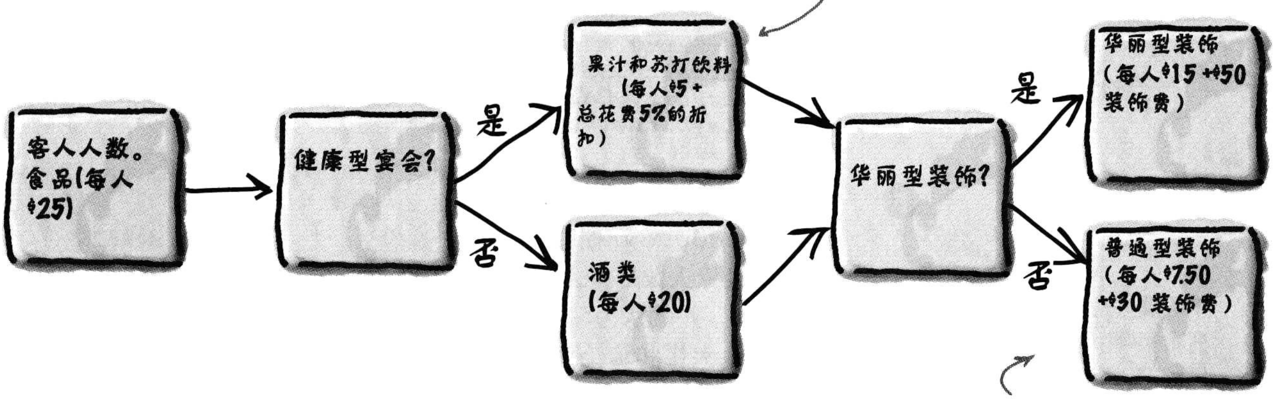
# 预算工具要做些什么？

Kathleen明确了计算活动花销的一些基本要点。以下是她提供的部分说明：

## Kathleen的聚会策划程序，一次宴会的花费预算

- 客人名单上的每个人食品花费为人均$25。
- 在饮料方面，客户可能有某种选择。大多数宴会都要提供酒类，人均花费为$20。不过也可能选择宴会不提供酒类。Kathleen称这种宴会为“健康型”宴会，只需为每个人提供苏打饮料和果汁，而不提供酒类，这样人均只需花费$5。选择“健康型”宴会对她来说筹划会容易得多，所以她还会给客户打一个折扣（整个宴会花费的5%）。
- 装饰品的花费有两种选择。如果客户只选择普通的装饰，则每人$7.50，另加$30的装饰费。客户还可以把宴会装饰升级为“华丽型”，这样每人将花费$15，另外还有$50的一次性装饰费。

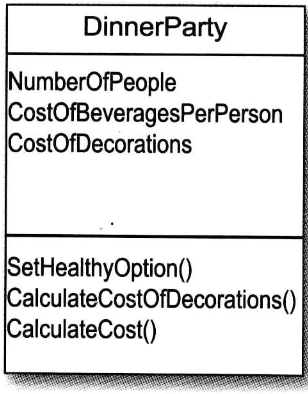
下面从另一个角度来看这些花费，我们把它转换成一个小流程图，以便你更清楚地了解这个过程：



**Exercise**

构建一个程序来解决Kathleen的宴会预算问题。

❶ 创建一个新的Windows Application工程，并增加一个类文件，名为`DinnerParty.cs`，使用左边的类图来构建这个DinnerParty类。它有3个方法：`CalculateCostOfDecorations()`、`SetHealthyOption()`和`CalculateCost()`。对于字段，两个有关花费的字段（CostOfBeveragesPerPerson和CostOfDecorations）使用decimal，人数（NumberOfPeople）使用int，记录是否选择健康型宴会的字段（HealthyOption）使用bool。要确保赋给decimal值的所有字面量后面都要增加一个M(10.0M)。




这是要创建的DinnerParty类的类图。

❷ 有一个很有用的C#工具。由于程序不会改变食品的人均花费，可以把它声明为一个常量（constant），常量与变量类似，只不过常量值永远不会改变。以下是这个常量的声明：

```
public const int CostOfFoodPerPerson = 25;
```

❸ 翻回到前一页，确保已经完全清楚这些方法的逻辑。其中只有一个方法有返回值（返回一个decimal），另外两个方法返回类型都是void。CalculateCostOfDecorations()方法根据参加宴会的人数计算出装饰花费。CalculateCost()方法把装饰花费与每个人的饮料和食品花费相加，得到总花费。如果客户选择“健康型”宴会，则可以在CalculateCost()中得出总花费之后再打折。

❹ 为窗体增加以下代码：

在窗体中声明dinnerParty字段，然后将这4行代码增加到InitializeComponent()下面。

```
DinnerParty dinnerParty;
public Form1() {
  InitializeComponent();
  dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
  dinnerParty.SetHealthyOption(false);
  dinnerParty.CalculateCostOfDecorations(true);
  DisplayDinnerPartyCost();
}
```

SetHealthyOption()方法使用一个bool参数(healthyOption)并根据客户是否选择健康型宴会来更新CostOfBeveragesPerPerson字段。

❺ 窗体应当右图所示。使用`NumericUpDown`控件的属性来设置人数最多为20，最少为1，并默认为5。还要去掉最大化和最小化按钮。

不需要为DinnerParty类增加“using System.Windows.Forms;”，因为它没有使用MessageBox.Show()或这个NET Framework命名空间的任何类。

设置默认值为5。最小值应当为1，最大值为20。

Fancy decorations复选框的Checked属性设置为true（选中）。




这是一个名为labelCost的标签，其Text属性为空（“”），BorderStyle属性设置为Fixed3D，另外AutoSize属性设置为false。

**6** 这个窗体没有使用按钮来计算花费，而是采用了另一种做法，一旦使用（单击）一个复选框（checkbox）或NumericUpDown控件，窗体就会自动地更新花费标签。首先需要在窗体中创建一个方法来显示这个花费。

向Form1()增加以下方法，点击NumericUpDown控件时会调用这个方法：

```
private void DisplayDinnerPartyCost()
{
    decimal Cost = dinnerParty.CalculateCost(checkBox2.Checked);
    costLabel.Text = Cost.ToString("c");
}
```

窗体上创建的所有其他方法都将调用这个方法。某个值发生变化时，都得通过这个方法将花费标签更新为正确的值。

为窗体增加这个方法，它会重新计算宴会的花费，并放在Cost标签中。

如果对应Healthy Option的复选框选中，这就为true。

将显示花费的标签改名为costLabel。

向ToString()传入"c"，这就告诉它将花费格式化为一个货币值。如果你所在的国家使用美元，就会增加一个美元符$。

**7** 现在将NumericUpDown域与DinnerParty类中创建的NumberOfPeople变量关联，并在窗体中显示花费。双击NumericUpDown控件，IDE会在代码中增加一个事件处理程序（event handler），这是每次控件发生改变时所要运行的一个方法。它会重置宴会人数。填入以下代码：

```
private void numericUpDown1_ValueChanged(
                                object sender, EventArgs e)
{
    dinnerParty.NumberOfPeople = (int) numericUpDown1.Value;
    DisplayDinnerPartyCost();
}
```

实际上你一直都在使用事件处理程序，双击一个按钮时，IDE就会增加一个单击事件处理程序（event handler）。现在你应该知道它的"大名"了。

需要将numericUpDown.Value强制转换为一个int，因为这是一个Decimal属性。

**唉呀！这个代码有一个问题。你能发现这个问题吗？如果现在看不出来也不必担心。稍后我们就会深入分析这个问题！**

从窗体发送到方法的值将是fancyBox.Checked。它会作为一个布尔参数传递到类中的方法。

这些方法只有两行代码。第一行调用类中为确定花费所创建的方法，第二行在窗体中显示这个总花费。

**8** 双击窗体上的Fancy Decorations复选框，确保首先调用CalculateCostOfDecorations()，然后调用DisplayDinnerPartyCost()。接下来，双击Healthy Option复选框，确保先调用DinnerParty类的SetHealthyOption()方法，然后调用DisplayDinnerPartyCost()方法。

**Exercise Solution**

以下是DinnerParty.cs的代码。

使用一个常量表示CostOfFoodPerPerson，确保这个值不会改变。这样还会使代码更可读，因为可以清楚地看出这个值永远不变。

窗体第一次创建这个对象时，它会使用初始化方法来设置NumberOfPeople。然后调用SetHealthyOption()和CalculateCostOfDecorations()设置其他字段。

```
class DinnerParty {
    const int CostOfFoodPerPerson = 25;
    public int NumberOfPeople;
    public decimal CostOfBeveragesPerPerson;
    public decimal CostOfDecorations = 0;

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption) {
            CostOfBeveragesPerPerson = 5.00M;
        } else {
            CostOfBeveragesPerPerson = 20.00M;
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        if (fancy)
        {
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        } else {
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
    }
    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = CostOfDecorations +
                ((CostOfBeveragesPerPerson + CostOfFoodPerPerson)
                    * NumberOfPeople);

        if (healthyOption) {
            return totalCost * .95M;
        } else {
            return totalCost;
        }
    }
}
```

这里使用了"if (Fancy)"而没有输入"if (Fancy == true)"，因为if语句总是检查条件是否为true。

这里使用小括号确保正确地完成数学计算。

如果选择了非酒精类（健康型），则对整个活动花费打5%的折扣。

这里对价格使用了decimal类型，因为它设计为存储货币值。一定要在每个字面量后面加一个“M”，因此，如果想存储$35.26，就要写为35.26M。这一点你肯定记得住，因为M就代表Money。

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty() { NumberOfPeople = 5 };
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void fancyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void healthyBox_CheckedChanged(object sender, EventArgs e) {
        dinnerParty.SetHealthyOption(healthyBox.Checked);
        DisplayDinnerPartyCost();
    }

    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
        DisplayDinnerPartyCost();
    }

    private void DisplayDinnerPartyCost() {
        decimal Cost = dinnerParty.CalculateCost(healthyBox.Checked);
        costLabel.Text = Cost.ToString("c");
    }
}
```

一旦窗体加载，就要调用DisplayDinnerPartyCost来初始化显示花费的标签。

对窗体上的复选框做出改变时，要在SetHealthyOption()和CalculateCostOfDecorations()方法中将healthyOption和Fancy布尔值设置为true或false。

将复选框命名为“healthyBox”和“fancyBox”，这样就可以看出相应的事件处理方法中要做什么。

只要人数改变或者选中了复选框，就要重新计算并显示新的宴会花费。

## 串格式化

你已经看到可以使用ToString()方法将任何变量转换为一个串。如果向ToString()传入“c”，就会把它转换为一个当地的货币值。还可以传入“f3”把它格式化为一个有3位小数的小数值；如果传入“0”（也就是一个零），则转换为一个整数；传入“0%”会转换为一个整数百分数；传入“n”会显示为一个有千位分隔符的数。花些时间来试试，看看程序中使用这些格式会有什么结果！

# Kathleen的测试




Rob是Kathleen最喜欢的客户之一。去年曾承办过他的婚礼，现在正准备为他举办一次宴会。

**Rob（电话中）**：嗨，Kathleen。我的宴会准备得怎么样了？

**Kathleen**：很好呀。我们今天早上刚去看了装饰材料，宴会一定会富丽堂皇，我想你肯定会喜欢的。

**Rob**：真令人期待。不过，我刚接到我妻子姨妈的电话。她和她丈夫想来待几个星期。你能告诉我，如果客人名单从10人改为12人的话，则预算会有多大的变化？

启动程序时，Fancy Decorations复选框应该已经选中，因为你将它的Checked属性设置为true。将人数设置为10会得到花费为$575。

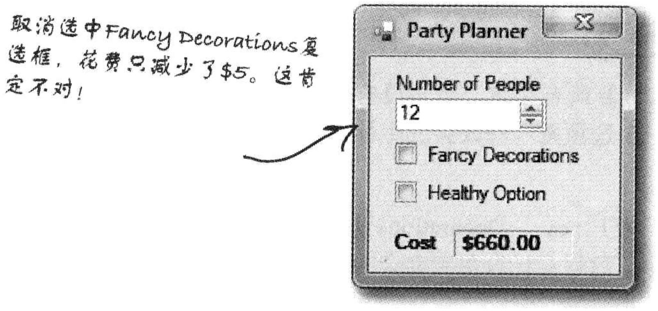
**Kathleen**：当然可以！则稍后给你答复。




把Number of People值从10改为12，再按Enter键，显示总花费为$665。嗯，好像有点低……

**Kathleen**：OK。看上去宴会的总花费会从$575增加到$665。

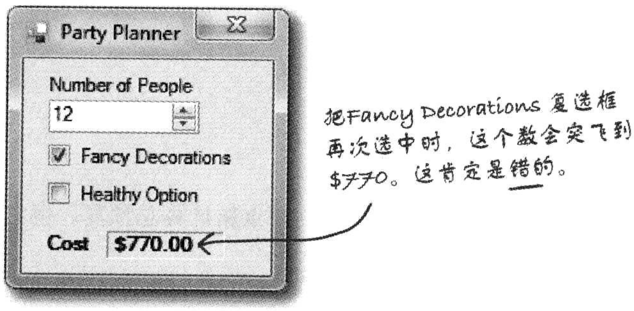
**Bob**：只差$90？听上去太值了。如果我们决定取消华丽装饰呢？花费又会是多少？




**Kathleen**：嗯，看上去……好像是$660。

**Rob**：$660？我想每个人的装饰费用应该是$15。你改了报价吗？还是改了别的什么？如果只差$5，那还是选择华丽装饰吧。不过，我要告诉你，这个报价真有些让人糊涂。

**Kathleen**：我们刚刚编写了这个新程序帮我们做预算。不过看起来可能还存在问题。稍等一会，我把华丽装饰再加回去。




**Kathleen**：Rob，我想这里有错误。看上去加了华丽装饰后花费涨到$770。这好像更合理一些。不过我已经不太相信这个应用了。我会让人修正里面的bug，另外自己动手来完成你的预算。我能明天再给你回复吗？

**Rob**：我可不会因为宴会只增加两个人就付$770。你先前说的价格更合适。我会付你最早说的$665，不过不能比这个更高了！

## BRAIN POWER

你觉得为什么每次Kathleen做出改变时会得到错误的结果？

# 每个选择都应当单独计算

尽管我们肯定是按照Kathleen所说的来计算所有花费，但是没有考虑这样一个问题，如果只对窗体上的某一个选项做了修改会发生什么情况。

启动这个程序时，窗体将人数设置为5，将Fancy Decorations设置为true（选中）。而Healthy Option未选中，这样会计算得到宴会的总花费为$350。初始的总花费是这样得到的：

> **Relax**
>
> **别担心！这不是你的错。**
>
> 在给你的代码中我们专门设置了一个讨厌的小bug，让你看看如果对象不加限制地使用彼此的字段多么容易出问题……另外要想找出这些问题将会多么困难。

**5个人时：**

每人饮料费用为$20。 → 饮料总花费 = $100

每人食品费用为$25。 → 食品总花费 = $125

每人装饰品费用为$15，外加$50的装饰费。 → 装饰总花费 = $125




到目前为止，一切正常。

$100 + $125 + $125 = $350

改变客人人数时，应用应该用同样的办法重新计算总预算。但它并没有这样做：

**10个人时：**

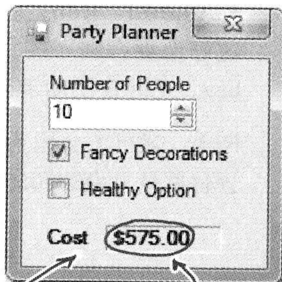
每人饮料费用为$20。 → 饮料总花费 = $200

每人食品费用为$25。 → 食品总花费 = $250

每人装饰品费用为$15，外加$50的装饰费。 → 装饰总花费 = $200

$200 + $250 + $200 = $650

这才是应该得到的总花费，但我们没有得到……




程序将原来的装饰费用增加到新的食品和饮料费用上。

它计算的是

$200 + $250 + $125 = $575

新的食品和饮料费用　　原来的装饰费用

**取消选中Fancy Decorations复选框，然后再将它选中。**

这会更新DinnerParty对象的CostOfDecorations字段，然后会显示正确的花费为$650。

## 问题放大镜

来看下面这个方法，numericUpDown控件中的值发生改变时要由这个方法做出处理。它把这个控件的值设置到NumberofPeople变量，再调用DisplayDinnerPartyCost()方法。然后依靠这个DisplayDinnerPartyCost()方法重新计算各个新的费用。

```
private void numericUpDown1_ValueChanged(
                                object sender, EventArgs e) {
   dinnerParty.NumberOfPeople = (int)numericUpDown1.Value;
   DisplayDinnerPartyCost();
}
```

这行代码将这个DinnerParty实例的NumberofPeople值设置为窗体中指定的值。

这个方法调用了CalculateCost()方法，但没有调用CalculateCostofDecorations()方法。

所以，改变NumberofPeople域中的值时，这个方法根本不会得到调用：

```
public void CalculateCostOfDecorations(bool Fancy) {
    if (Fancy) {
        CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
    } else {
        CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
    }
}
```

这个变量在窗体第一次调用这个方法时设置为$125，由于不会再调用该方法，所以这个变量不再改变。

正是因为这个原因，将fancy decorations复选框再次选中时又会得到正确的得数。单击这个复选框时，程序会再次运行CalculateCostOfDecorations()。



等一等！我以为Kathleen总是会同时设置所有这3个选项！

人们往往不会按你期望的方式来使用你的程序。幸运的是，C#提供了一个强大的工具，可以确保你的程序总能正确地工作，即使人们做了你不曾预料的事情也能很好地应对。这个工具称为封装（encapsulation），这确实是处理对象的一个非常有用的技术。

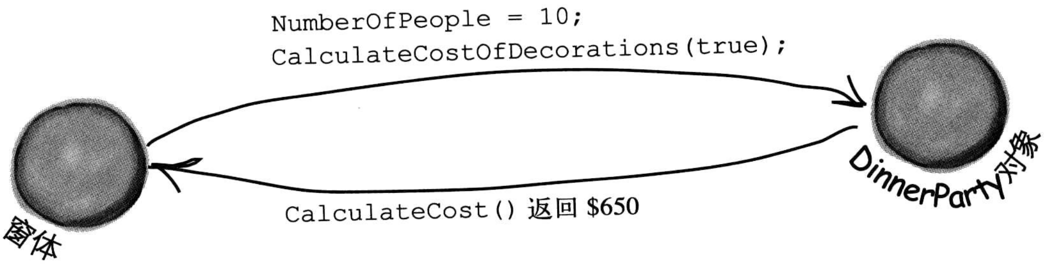
……而且有时想使用这些类的“人”就是你自己！你今天写的类可能会在明天使用。

# 很容易无意识地滥用对象

Kathleen之所以会遇到麻烦，主要是因为她的窗体没有用你已经建立的便利方法CalculateCostOfDecorations()，而是直接去访问DinnerParty类中的字段。所以，尽管你的DinnerParty类能正常工作，但窗体采用了一种意料之外的方式来调用它……这就带来了问题。

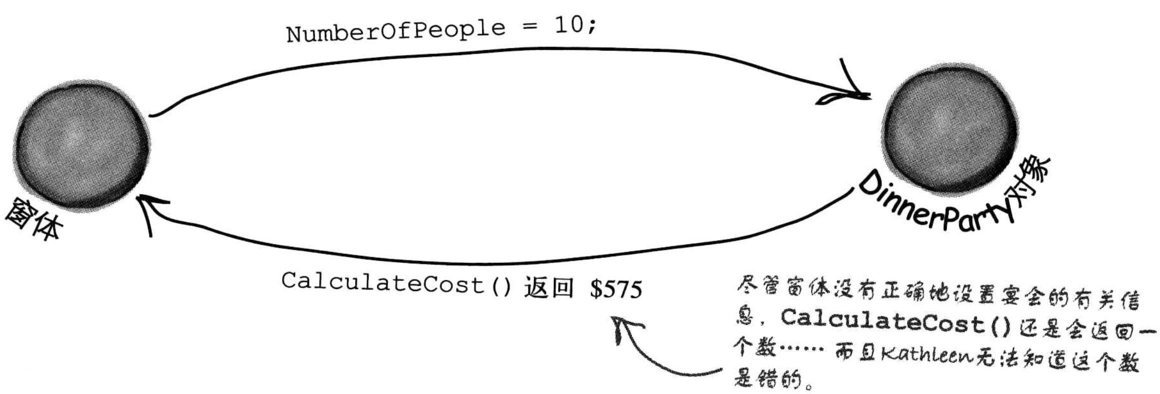
**❶ 应该怎么调用DinnerParty类。**

DinnerParty类为窗体提供了一个非常好的方法来计算装饰的总花费。只需要设置人数，再调用CalculateCostOfDecorations()，然后CalculateCost()就会返回正确的花销。




**❷ 实际上DinnerParty类是如何调用的。**

窗体设置了人数，但是只调用了CalculateCost()方法而没有首先重新计算装饰费用。这就把整个计算都搞砸了，以至于Kathleen最后向Rob给出了错误的报价。



# 封装意味着保证类中的一些数据是私有的

要避免这种问题有一种很简单的方法：确保只能通过一种途径使用这个类。幸运的是，C#允许将某些字段声明为私有（privite），因此很容易做到这一点。到目前为止，你所看到的都只是一些公共（public）字段，所有其他对象都可以读取或修改这种字段。不过，如果把它设置为一个私有字段，就只能在这个对象内部（或者由同一个类的另一个对象）访问该字段。

充分考虑到你的懒惰，如果没有加上"private"或"public"，C#就会认为这个字段是私有的。

另外一个类的静态方法也可以访问该类任何实例的私有字段。

```
class DinnerParty {
    private int numberOfPeople;

    ...

    public void SetPartyOptions(int people, bool fancy) {
        numberOfPeople = people;
        CalculateCostOfDecorations(fancy);
    }

    public int GetNumberOfPeople() {
        return numberOfPeople;
    }
```

如果希望将一个字段设置为私有，只需在声明字段时使用private关键字。这就会告诉C#，如果有innerParty的一个实例，就只能由这个实例读/写其numberOfPeople字段。其他对象甚至都不知道这个字段的存在。

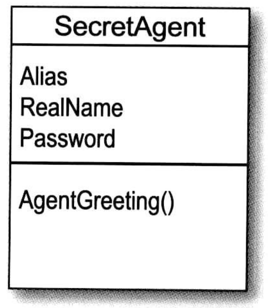
还需要为其他对象提供一个途径来设置宴会的人数。要访问这个字段，一个很好的办法是增加设置或获取人数的方法。这样一来，可以确保每次人数发生改变时都会运行CalculateCostOfDecorations()方法。这样就能消除这个讨厌的bug。

**将保存宴会人数的字段置为private，从而只为窗体提供一个途径来告诉DinnerParty类宴会中会有多少人参加，而且可以确保能正确地重新计算装饰费用。将某些数据置为私有并编写代码来使用这个数据，这就称为封装（encapsulation）。**

**en–cap–su–la–ted，封装（封闭），形容词。**
由一个保护性的外壳或膜包起来。潜水员们完全封闭在潜艇里，只能通过密封过渡仓出入。

# 使用封装来控制对类方法和字段的访问

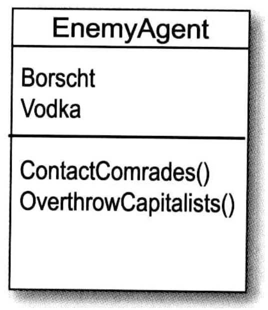
将所有字段和方法都置为公共时，所有其他类都将能够访问。类所做的一切以及知道的一切对于程序中的其他类来说将一览无余……前面已经看到，这将导致程序有意想不到的表现。利用封装，允许你控制类中的哪些部分可以分享，而哪些必须留作私有。下面来看这是如何做到的：




1. 超级密探Herb Jones是潜伏在某国（WHAT）的一个秘密间谍，要保住自己的生命安全、人身自由和幸福。他的`ciaAgent`对象是`SecretAgent`类的一个实例。

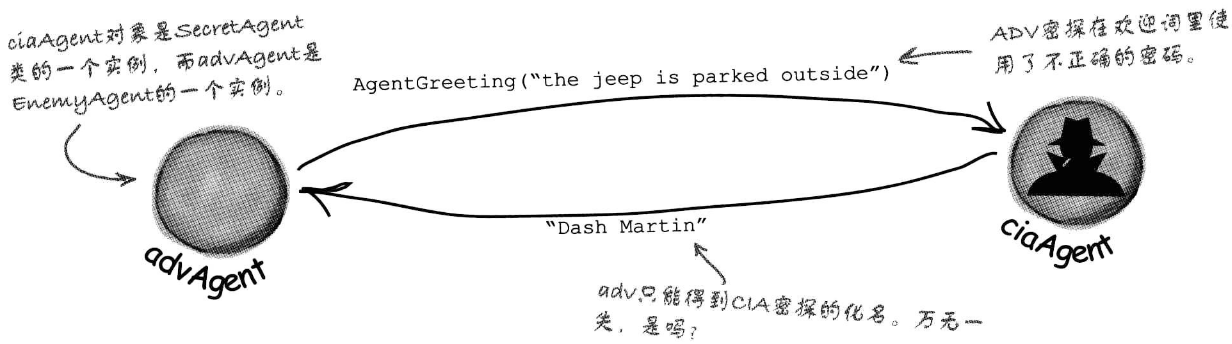



```
RealName: "Herb Jones"
Alias: "Dash Martin"
Password: "the crow flies at midnight"
```




2. 密探Jones有一个计划来帮助他躲过敌方的高级密探（ADV）密探。他增加了一个`AgentGreeting()`方法，以一个密码作为参数。如果没有得到正确的密码，就只给出他的化名Dash Martin。

3. 看上去是一个万无一失的办法，完全可以保护这个密探的身份，是这样吗？只要调用这个方法的密探对象没有正确的密码，Jones密探的名字就是安全的。

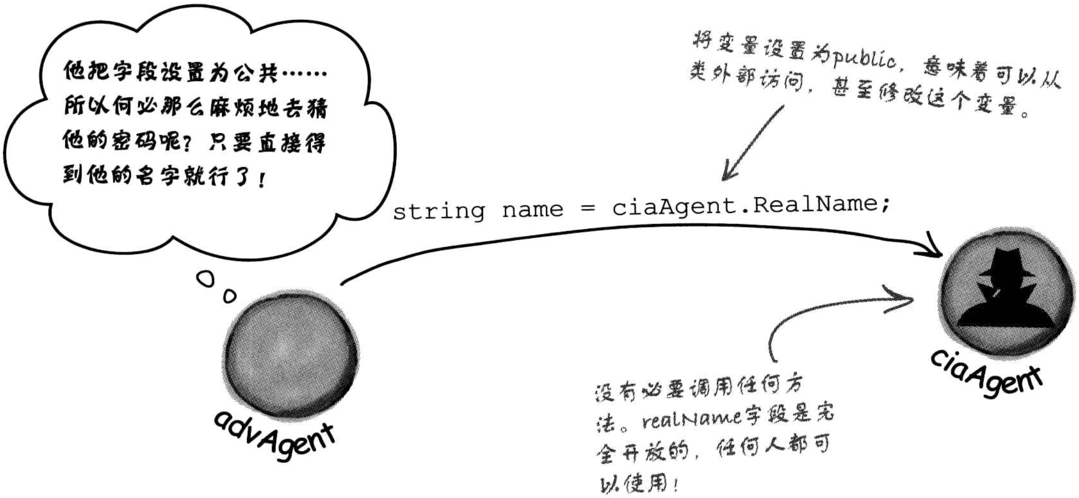


# 但是realName字段真的得到了保护吗？

所以，只要ADV不知道CIA密探的密码，CIA的真实姓名就是安全的。是这样吗？但是如果realName字段的声明如下会怎么样呢？

将变量设置为public，意味着可以从类外部访问，甚至修改这个变量。

```
public string RealName;
```




密探Jones可以使用**private**字段来保证他的身份不被敌方密探对象识破。一旦将realName字段声明为私有，得到这个真实名字的唯一途径就是调用能够访问类中私有部分的方法。所以ADV密探的企图将成为泡影！

advAgent对象不能访问ciaAgent的私有字段，因为advAgent对象不是SecretAgent类的实例。

只需将public替换为private，砰，你的字段会立即从这个世界隐身。

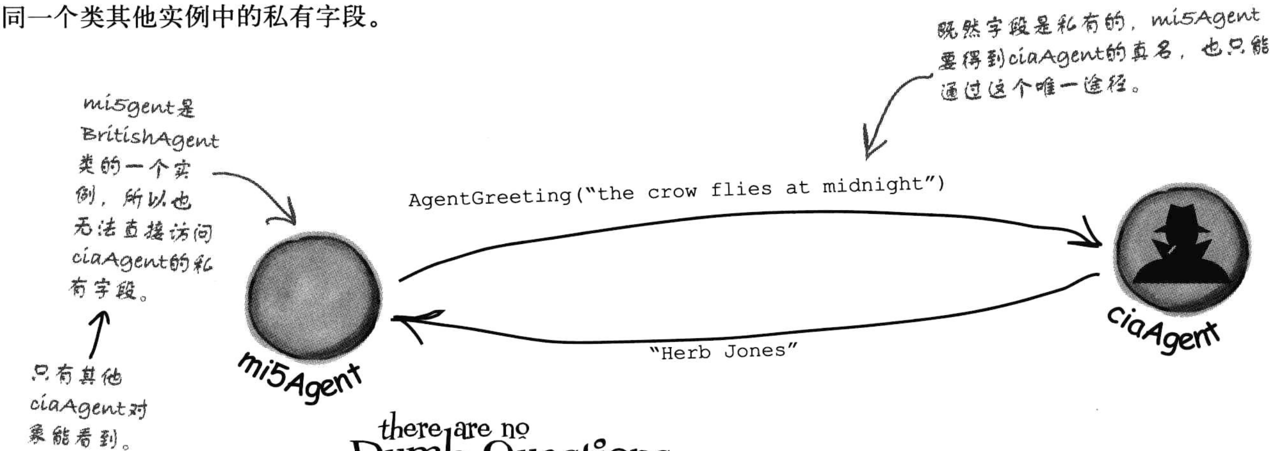
```
private string realName;
```

保证字段和方法私有，可以确保外部代码无法在你不期望的情况下修改你使用的值。

你可能还希望保证存放密码的字段也是私有的，否则，敌方密探就可以得到这个密码。

# 私有字段和方法只能从类的内部访问

一个对象要得到另一个对象私有字段中存储的数据，只有一个办法：就是使用返回该数据的公共字段和方法。ADV和MI5密探都需要使用AgentGreeting()方法，但友方密探可以看到所有信息。任何类都可以看到同一个类其他实例中的私有字段。




## there are no Dumb Questions

**问：那好，这么说我需要通过公共方法来访问私有数据。但是，如果包含私有字段的类没有提供获取这个数据的方法，而我的对象确实需要使用这个数据，会怎么样呢？**

**答：** 如果是这样，则将无法从这个对象外部访问这个数据。写一个类时，一定要保证为其他对象提供了某个途径来得到它们需要的数据。私有字段是封装的一个很重要的部分，但是它们并不是问题的全部。要编写一个封装性很好的类，意味着要为其他对象提供一个合理的、易于使用的途径来得到它们需要的数据，而不允许它们非法截获你的类本身需要的数据。

**问：既然其他类无法访问，为什么还要保留这样一个字段呢？**

**答：** 有时一个类需要记录一些必要的信息来完成操作，但是其他对象确实不需要看到这些信息。下面举一个例子。计算机生成随机数时，会使用一些特殊的值，称为种子（seed）。你不需要知道这是如何工作的，但是要知道Random的每个实例实际上包含一个数组，其中有数十个数，Random使用这个数组来确保Next()总会提供一个随机数。创建Random的一个实例时，不会看到这个数组。这是因为你根本不需要它，倘若你能访问这个数组，就可以在其中放入一些值，这会导致生成非随机的值。所以种子对你完全是封装的。

**问：嘿，我刚注意到，我用的所有事件处理程序都有private关键字。为什么它们都是私有的？**

**答：** 因为C#窗体要求只能由窗体上的控件触发事件处理程序。在方法前放上private关键字时，这个方法就只能在类内部使用。IDE向程序增加一个事件处理方法时，会把它声明为私有，使得其他窗体或对象无法调用这个方法。但是并没有明确的规定要求事件处理程序必须是私有的。实际上，你可以自己做一个检查，双击一个按钮，然后把它的事件处理程序声明改为public。现在代码应该仍能编译并运行。

**一个对象要得到另一个对象私有字段中存储的数据，唯一的途径就是使用返回该数据的公共方法。**

## Sharpen your pencil

以下类包含一些私有字段。如果在类外部使用一个名为**mySuperChef**的对象实例运行下面的语句，则有些语句将无法编译，将这些语句圈出来。

```
class SuperChef
{
      public string cookieRecipe;
      private string secretIngredient;
      private const int loyalCustomerOrderAmount = 60;
      public int Temperature;
      private string ingredientSupplier;

      public string GetRecipe (int orderAmount)
      {
            if (orderAmount >= loyalCustomerOrderAmount)
            {
                  return cookieRecipe + " " + secretIngredient;
            }
            else
            {
                  return cookieRecipe;
            }
      }
}
```

1. `string ovenTemp = mySuperChef.Temperature;`

2. `string supplier = mySuperChef.ingredientSupplier;`

3. `int loyalCustomerOrderAmount = 94;`

4. `mySuperChef.secretIngredient = "cardamom";`

5. `mySuperChef.cookieRecipe = "get 3 eggs, 2 1/2 cup flour, 1 tsp salt, 1 tsp vanilla and 1.5 cups sugar and mix them together. Bake for 10 minutes at 375. Yum!";`

6. `string recipe = mySuperChef.GetRecipe(56);`

7. 运行以上所有能编译的代码之后，recipe的值是什么？

.................................................................................................

.................................................................................................

## Sharpen your pencil Solution

以下类包含一些私有字段。如果在类外部使用一个名为**mySuperChef**的对象实例运行下面的语句，则有些语句将无法编译，将这些语句圈出来。

```
class SuperChef
{
      public string cookieRecipe;
      private string secretIngredient;
      private const int loyalCustomerOrderAmount = 60;
      public int Temperature;
      private string ingredientSupplier;

      public string GetRecipe (int orderAmount)
      {
            if (orderAmount >= loyalCustomerOrderAmount)
            {
                  return cookieRecipe + " " + secretIngredient;
            }
            else
            {
                  return cookieRecipe;
            }
      }
}
```

得到秘密配方的唯一途径是订购大量饼干。外部代码不能直接访问这个字段。

1. `string ovenTemp = mySuperChef.Temperature;`

#1不能编译，因为不能把一个int赋给一个string。

2. `string supplier = mySuperChef.ingredientSupplier;`

3. `int loyalCustomerOrderAmount = 54;`

#2和#4不能编译，因为ingredientSupplier和secretIngredient是私有的。

4. `mySuperChef.secretIngredient = "cardamom";`

5. `mySuperChef.cookieRecipe = "Get 3 eggs, 2 1/2 cup flour, 1 tsp salt, 1 tsp vanilla and 1.5 cups sugar and mix them together. Bake for 10 minutes at 375. Yum!";`

6. `string recipe = mySuperChef.GetRecipe(56);`

尽管创建了一个局部变量loyalCustomerAmount，并设置为54，但这不会改变对象的loyalCustomerAmount值，它仍为60，所以不会打印出秘密配方。

7. 运行以上所有能编译的代码之后，recipe的值是什么？

"Get 3 eggs, 2 1/2 cup flour, 1 tsp salt, 1 tsp vanilla and 1.5 cups sugar and mix them together. Bake for 10 minutes at 375. Yum!"




因为有时你希望类能够向程序的其余部分隐藏信息。很多人第一次见到封装时都觉得它有点奇怪，因为隐藏一个类的字段、属性或方法的想法有些违背常理。不过，确实有一些很充分的理由促使你考虑类中的哪些信息可以向程序的其余部分公开。

## 封装可以使你的类……

★ **易于使用。**

你已经知道了类使用字段来跟踪其状态。另外，很多类都使用方法来保证这些字段反映最新状态，其他类根本不会调用这些方法。一个类可能包含从未由任何其他类调用的字段、方法和属性，这种情况很常见。如果将这些成员置为私有，需要使用这个类时，这些成员就不会在智能提示窗口中显示出来。

★ **易于维护。**

还记得Kathleen程序中的bug吗？之所以会发生那种情况，是因为窗体直接访问了一个字段，而不是使用一个方法来设置它。如果这个字段是私有的，就完全可以避免这个bug。

★ **灵活。**

很多情况下，你可能希望退一步，向之前写的某个程序增加一些特性。如果你的类得到了很好的封装，以后你就能清楚地知道该如何使用。

**封装是指让一个类对另一个类隐藏信息。这有助于避免程序中出现bug。**

**BRAIN POWER**

如果现在构建一个封装性差的类，为什么以后会导致程序更难修改？

## Mike的导航程序可以使用更好的封装

定向追踪（Geocaching）是一项运动，人们使用GPS导航仪隐藏和查找容器，它们可以藏在世界上任何位置。Mike确实对GPS很在行，这样你就知道为什么他对这项运动如此着迷了吧。

还记得第3章中Mike的街道导航程序吗？现在Mike加入了一个定向追踪小组，他觉得他的导航程序可以让他技高一筹。不过从他开始使用至今已经过去了一段时日，现在他遇到了一个小麻烦。Mike的导航程序有一个`Route`类，用来存储两点之间的一条路线。不过他屡屡遇到各种bug，因为他看起来无法确定如何使用！Mike试图再来看他的导航程序，修改代码，此时发生了以下问题：

- ★ Mike将`StartPoint`属性设置为他家的GPS坐标，并把`EndPoint`属性设置为他办公室的坐标，并检查了`Length`属性。这个属性指出长度为15.3。但是调用`GetRouteLength()`方法时，居然返回0。
- ★ 他使用`SetStartPoint()`属性将起点设置为他家的坐标，使用`SetEndPoint()`属性将终点设置为他的办公室。`GetRouteLength()`方法返回9.51，而`Length`属性包含的值为5.91。
- ★ 他试图使用`StartPoint`属性设置起点和`SetEndPoint()`方法设置终点时，`GetRouteLength()`总是返回0，而且Length属性也总是包含0。
- ★ 他试图使用`SetStartPoint()`方法设置起点，而使用`EndPoint`属性设置终点时，`Length`属性包含0，而且`GetRouteLength()`方法会导致程序因一个错误而崩溃，错误的内容与无法除0有关。




### Sharpen your pencil

下面给出Mike导航程序中的Route对象。你认为哪些属性或方法应该置为私有会让它更易于使用？

| Route |
| --- |
| StartPoint<br>EndPoint<br>Length |
| GetRouteLength()<br>GetStartPoint()<br>GetEndPoint()<br>SetStartPoint()<br>SetEndPoint()<br>ChangeStartPoint()<br>ChangeEndPoint() |

..................................................................

..................................................................

..................................................................

..................................................................

..................................................................

..................................................................

解决这个问题有很多种方法，可能所有这些方法都是正确的！请写出你认为最好的方法。

## 可以把对象认为是一个黑箱

有时你会听到有程序员把对象称为一个“黑箱”，这是一种很不错的考虑方式。调用一个对象的方法时，你并不关心这个方法是如何工作的。至少，现在不关心。你关心的只是它要取你提供的输入，并返回适当的结果。

**如果再回头看你很长时间都没有看的代码，很容易忘记原来打算用它们来做什么。在这里利用封装可以让你的日子好过得多！**

我知道我的Route对象能正常工作！对我来说，现在我想知道如何把它用于我的定向追踪工程。

回顾第3章，Mike在考虑如何构建他的导航程序。那时他确实很关心Route对象如何工作。不过那已经过去一段时间了。

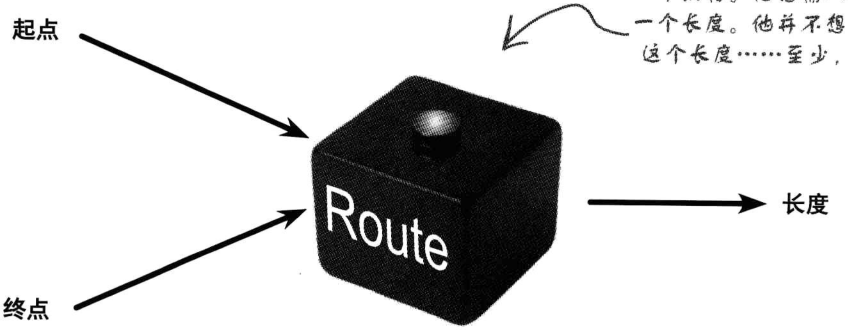
在那之后，他已经让导航程序运行起来，而且已经使用了相当长的一段时间。他知道这个导航程序可以很好地工作，肯定对他的定向追踪小组有帮助。现在他希望重用他的Route对象。

**如果你今天能很好地封装类，明天重用这些类会容易得多。**



如果Mike原先构建Route对象时能想到封装就好了！如果他能想到，现在就不会这么头疼了！

现在，Mike只想把他的Route对象考虑为一个黑箱。他想输入他的坐标，从中得到一个长度。他并不想考虑Route如何计算这个长度……至少，现在不想。



这么说，一个封装得好的类与封装得不好的类做的工作是一样的！



**完全正确！区别在于，封装得好的类采用一种可以避免bug的方式构建，而且更易于使用。**

很容易把一个封装性好的类变成一个封装性差的类：只需要做一个查找，将`private`的每一个出现都替换为`public`。

关于`private`关键字有一个有趣的现象：你通常可以对任何程序完成这种查找替换，它仍能正常编译，而且以同样的方式完成工作。这也是一些程序员很难理解封装的一个原因。

直到现在，你学到的所有知识都是让程序做某些事情——完成某些行为。封装稍有不同。它并不会改变程序的行为。它更多地有关于编程的“象棋博弈”部分：通过在设计和构建类时隐藏某些信息，可以为以后类之间如何交互建立一个策略。这个策略越好，你的程序就越灵活，而且更可维护，相应地可以避免更多的bug。

就像象棋一样，可能的封装策略几乎是无限的！

# 实现类封装的几个主意

### ★ 考虑字段可能以何种方式被滥用。

如果设置不当，则会出什么问题？

### ★ 是不是类中的所有一切都是公共的？

如果类中只有公共字段和方法，则可能需要多花一些时间来考虑封装问题。

### ★ 哪些字段需要在设置时做一些处理或计算？

这些是最有可能需要封装的字段。如果有人以后写了一个方法来改变其中某个字段的值，则可能会带来问题，导致你的程序无法完成工作。




### ★ 只将必要的字段和方法声明为公共。

如果没有适当的原因，就不要将字段或方法声明为公共。如果程序中的所有字段都声明为公共字段，则可能会把问题搞得一团糟。不过也不要把一切都设置为私有。先花一些时间来考虑这个问题，哪些字段确实需要是公共的，而哪些不必，这会为你在以后节省很多时间。

# 封装保证数据干净

有时程序正常完成工作时字段中的值会改变。如果没有明确地告诉程序重新设置这个值，就会使用原来的值完成计算。对于这种情况，你可能希望一旦字段改变就让程序执行一些语句，如每次改变人数时就让Kathleen的程序重新计算花费。通过使用私有字段封装数据，可以避免这个问题。我们将提供一个方法得到这个字段的值，还会提供另一个方法来设置这个字段，并完成所有必要的计算。

## 封装的一个小例子

一个Farmer类使用一个字段存储奶牛数量，把奶牛数乘以某个数来得出喂养这些奶牛需要多少包饲料：

```
class Farmer
{
      private int numberOfCows;
}
```

最好设置这个字段为私有字段，这样一来，如果不同时修改bagsOfFeed，就无法修改这个字段。如果这两个数不同步，肯定会有bug！

创建一个窗体，允许用户在一个数字域中输入奶牛数量，需要能够修改numberOfCows字段中的值。为此，可以创建一个方法向窗体对象返回这个字段的值。

```
public const int FeedMultiplier = 30;
public int GetNumberOfCows()
{
      return numberOfCows;
}

public void SetNumberOfCows(int newNumberOfCows)
{
      numberOfCows = newNumberOfCows;
      BagsOfFeed = numberOfCows * FeedMultiplier;
}
```

农夫需要为每头奶牛准备30包饲料。

增加一个方法，为其他类提供一个途径来获得奶牛数。

完成的工作是一样的！

私有字段采用camelCase风格，公共字段采用PascalCase风格。PascalCase表示变量名中每个单词的首字母大写；camelCase与PascalCase类似，不过第一个字母小写。这样会使大写字母看上去像是骆驼（camel）的“驼峰”。

这个方法用于设置奶牛数，并确保会同时调整BagsOfFeed字段。现在这两个字段绝对同步。

# 属性使封装更容易

可以使用属性（properties），这些方法对其他对象来说就像是字段。可以用属性来获取或设置一个后备字段（（backing field），后备字段就是由属性所设置的一个字段的名。

```
private int numberOfCows;

public int NumberOfCows
{
    get
    {
        return numberOfCows;
    }
    set
    {
        numberOfCows = value;
        BagsOfFeed = numberOfCows * FeedMultiplier;
    }
}
```

将这个私有字段重命名为numberOfCows（注意小写字母"n"）。它将成为NumberOfCows属性的后备字段。

使用属性时要结合一个正常的字段声明。以下是NumberOfCows的声明。

这是一个获取存取方法（get accessor）。每次读取NumberOfCows字段时都会运行这个方法。它的返回值类型与这个变量类型匹配。在这里它会返回私有字段numberOfCows的值。

这是一个设置存取方法（set accessor），每次设置NumberOfCows字段时会调用这个方法。尽管这个方法看上去没有任何参数，但实际上它有一个名为value的参数，其中包含为这个字段设置的值。

使用获取和设置存取方法与使用字段非常类似。以下是一个按钮的代码，它将设置奶牛数，然后得到饲料包数：

```
private void button1_Click(object sender, EventArgs e) {
    Farmer myFarmer = new Farmer();
    myFarmer.NumberOfCows = 10;

    int howManyBags = myFarmer.BagsOfFeed;

    myFarmer.NumberOfCows = 20;
    howManyBags = myFarmer.BagsOfFeed;
}
```

这一行将NumberOfCows设置为10时，设置存取方法将设置私有numberOfCows字段，然后更新公共字段BagsOfFeed。

由于NumberOfCows设置存取方法更新了BagsOfFeed，现在可以得到它的值。

尽管代码把NumberOfCows看做是一个字段，但实际上它会运行设置存取方法，并传入20。查询BagsOfFeed字段时，会运行获取存取方法，这会返回300。

# 构建一个应用测试Farmer类

创建一个新Windows Forms应用，可以用它来测试**Farmer**类，并查看属性的具体工作。我们将使用**Console.WriteLine()**方法将结果写到IDE中的输出窗口。

**动手做！**

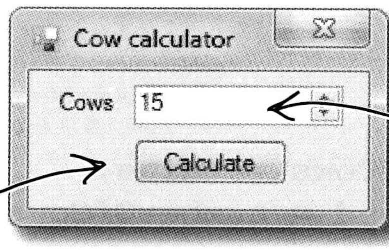
**1** 为工程增加Farmer类。

```
public class Farmer {
    public int BagsOfFeed;
    public const int FeedMultiplier = 30;

    private int numberOfCows;
    public int NumberOfCows {
        (增加上一页的获取和设置存取方法)
    }
}
```

**2** 建立这个窗体：

将这个按钮命名为“calculate”，它使用公共Farmer数据向输出窗口写一行文本。




将NumericUpDown控件的Value设置为15，设置Minimum为5，Maximum设置为300。

**3** 以下是窗体代码。这里使用Console.WriteLine()将输出发送到Output窗口（从“Debug”的“Windows”菜单选择“Output”就会显示这个窗口）。可以向WriteLine()传递多个参数，第一个参数是要写出的串。如果在这个串中包含一个“{0}”，则WriteLine()将把它替换为第1个参数，“{1}”会替换为第2个参数，“{2}”替换为第3个参数，以此类推。

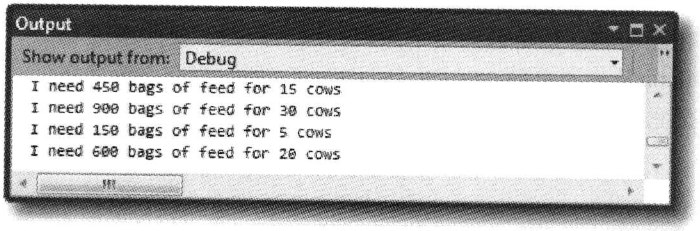
```
public partial class Form1 : Form {
    Farmer farmer;
    public Form1() {
        InitializeComponent();
        farmer = new Farmer() { NumberOfCows = 15 };
    }
    private void numericUpDown1_ValueChanged(object sender, EventArgs e) {
        farmer.NumberOfCows = (int)numericUpDown1.Value;
    }
    private void calculate_Click(object sender, EventArgs e) {
        Console.WriteLine("I need {0} bags of feed for {1} cows",
            farmer.BagsOfFeed, farmer.NumberOfCows);
    }
}
```

使用Console.WriteLine()方法将一行文本发送到IDE的Output窗口。

WriteLine()将“{0}”替换为第1个参数中的值，“{1}”替换为第2个参数。

# 使用自动属性完成这个类

看起来Cow Calculator干得不错。来试一试，运行这个程序，单击按钮。然后将奶牛数改为30，再单击按钮。接下来再把奶牛数改为5，然后改为20。Output窗口的输出应该如下：




能看出来吗？这可能会让你无意中在程序中增加一个相当烦人的bug。

不过这个类还有一个问题。向窗体增加一个按钮，执行以下语句：

```
farmer.BagsOfFeed = 5;
```

现在再来运行这个程序。单击新按钮之前它都能很好地工作。不过，按下这个按钮，然后再单击Calculate按钮。现在输出中会指出需要5包饲料，不论你有多少头奶牛都只需要5包饲料！一旦改变NumericUpDown，Calculate按钮又能正常工作了。

## 完全封装Farmer类

问题出在这个类没有完全封装。这里使用了属性来封装NumberOfCows，但是BagsOfFeed还是公共的。这是一个很常见的问题。实际上，正是因为这个问题如此常见，C#提供了一种方法来自动进行修正。只需将公共字段BagsOfFeed改为一个自动属性（automatic property）。利用IDE可以很容易地增加自动属性。步骤如下：

输入prop，再按两次Tab，就会向代码增加一个自动属性。

1. 从Farmer类去掉BagsOfFeed字段。把光标停在这个字段原来的位置上，然后输入**prop**，再按两次Tab键。IDE会增加下面这行代码：

```
public int MyProperty { get; set; }
```

2. 接下Tab键，光标会跳到MyProperty。把这个名字改为BagsOfFeed：

```
public int BagsOfFeed { get; set; }
```

现在就得到了一个属性而不是字段。C#看到这个属性时，就好像你使用了一个后备字段一样（就像公共NumberOfCows属性后面有一个私有的numberOfCows字段）。

3. 问题还没有解决。不过很容易修正，只需把它设置为一个只读属性（read-only property）：

```
public int BagsOfFeed { get; private set; }
```

重新构建这个代码，设置BagsOfFeed的按钮代码行中会出现一个错误，指出设置存取方法不可访问。不能在类之外修改BagsOfFeed，需要将这行代码删除才能正常编译，所以将这个按钮从窗体删除。现在Farmer类得到了更好的封装！

# 如果想改变单位饲料数呢？

动手做！

我们已经构建了Cow Calculator，其中对每头奶牛的饲料数（单位饲料数）使用了一个const。但是，如果我们希望在其他程序中同样使用这个Farmer类，不过需要不同的单位饲料数，该怎么做呢？你已经看到，如果封装不当，则可能导致出现问题，以至于一个类中的字段在其他类中也能访问。正是因为这个原因，应当只把必要的字段和方法置为公共。由于Cow Calculator从来不更新FeedMultiplier，所以没有必要允许其他类设置这个值。因此下面把它改为一个使用后备字段的只读属性。

**1** 从程序中去掉这一行：

```
public const int FeedMultiplier = 30;
```

使用prop-tab-tab片段增加一个只读属性。不过不是增加一个自动属性，这里要使用一个后备字段：

```
    private int feedMultiplier;
    public int FeedMultiplier { get { return feedMultiplier; } }
}
```

这个属性就相当于一个int字段，只不过并非存储一个值，它只是返回后备字段feedMultiplier。由于没有设置存取方法，所以这是只读属性。它有一个公共的获取存取方法，这说明任何其他类都可以读取FeedMultiplier的值。但是由于其设置存取方法是私有的，因此它是只读的，只能由Farmer的实例来设置。

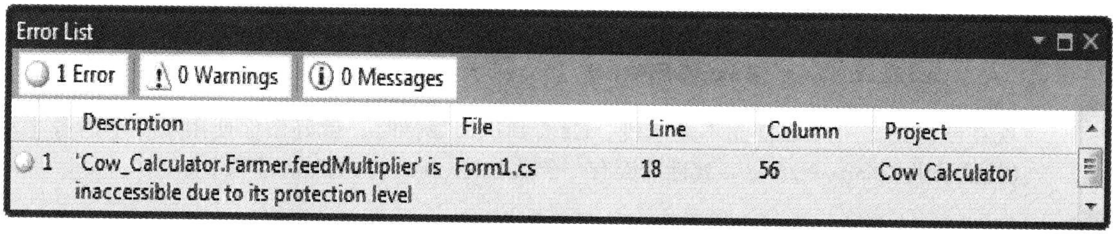
由于我们将FeedMultiplier从一个公共的常量（public const）改为一个私有的int字段，所以要改名，以一个小写的"f"开头。这是一种相当标准的命名约定，本书中将一直采用这种约定。

**2** 修改代码后运行程序。唉呀，出问题了！BagsOfFeed总返回0包！

等一下，这是有道理的。FeedMultiplier一直都没有初始化。它开始的默认值是0，然后从未改变过。把它乘以奶牛数时，肯定还会得到0。所以下面增加一个对象初始化方法：

```
public Form1() {
    InitializeComponent();
    farmer = new Farmer() { NumberOfCows = 15, feedMultiplier = 30 };
```

唉呀，程序无法编译！你会得到下面这个错误：




**对象初始化方法中只能初始化公共字段和属性。所以如果需要初始化的一些字段是私有字段，如何保证对象正确地初始化呢？**

# 使用构造函数初始化私有字段

如果需要初始化对象，不过需要初始化的一些字段是私有字段，那么就不能使用对象初始化方法。幸运的是，任何类都可以增加一个特殊的方法，称为构造函数（constructor）。如果类有一个构造函数，用new语句创建类时首先就会运行这个构造函数。可以向构造函数传递参数，为需要初始化的字段指定值。不过构造函数没有返回值，因为我们不会直接调用这个方法。要把它的参数传入new语句。你已经知道了，new会返回对象，所以构造函数不会返回任何结果。

**要向类增加一个构造函数，只需增加一个与类同名的方法，而且没有返回值。**

**1** **为Farmer类增加一个构造函数。**

这个构造函数只有两行代码，不过完成的工作可不少。下面来一步一步地分析。我们已经知道，这个类需要有奶牛数和单位饲料数，所以把它们作为构造函数的参数。由于我们将`feedMultiplier`从一个`const`改为一个`int`，现在需要为它指定一个初始值。所以下面确保这个值传入构造函数。我们还将使用构造函数来设置奶牛数。

注意这里在“public”后面没有“void”或“int”之类的类型，这是因为构造函数没有返回值。

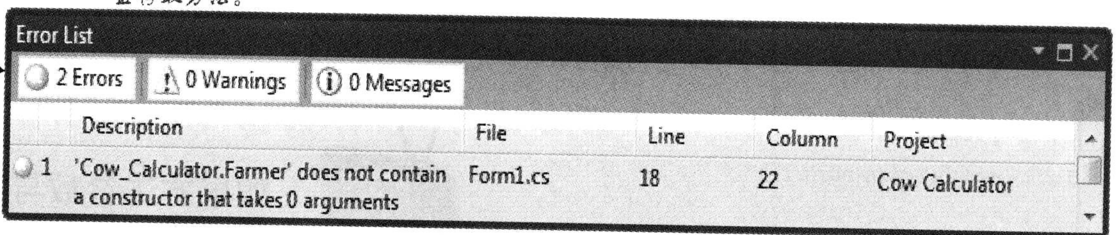
this.feedMultiplier中的“this”关键字告诉C#你指的是字段，而不是同名的参数。

```
public Farmer(int numberOfCows, int feedMultiplier) {
    this.feedMultiplier = feedMultiplier;
    NumberOfCows = numberOfCows;
}
```

首先要做的是设置单位饲料数，因为这要在调用NumberOfCows设置存取方法之前设置。

如果只是设置私有字段numberOfCows，NumberOfCows设置存取方法就将永远也得不到调用。设置this.NumberOfCows可以确保调用这个设置存取方法。

如果构造函数需要参数，而new()没有提供任何参数，就会得到这个错误。




**2** **现在修改窗体来使用构造函数。**

现在只需要修改窗体，使创建`Farmer`对象的new语句使用这个构造函数而不是对象初始化方法。一旦替换new语句，两个错误都会消失，代码将正常运行！

```
public Form1() {
    InitializeComponent();
    farmer = new Farmer(15, 30);
}
```

你已经知道，窗体是一个对象。没错，它也有一个构造函数！这正是这个构造函数，注意它名为Form1（与类同名），而且没有返回值。

在这里new语句调用了构造函数。看上去与其他new语句完全一样，只不过这里向构造函数方法传入了参数。输入这行代码时，注意智能提示的弹出窗口，看起来与所有其他方法都一样。

## 构造函数放大镜

下面更仔细地分析Farmer构造函数，搞清楚具体是怎样做的。

构造函数不返回任何结果，所以没有返回类型。

这个构造函数有两个参数，与平常的参数一样。第一个参数指定了奶牛数，第二个参数是单位饲料数。

```
public Farmer(int numberOfCows, int feedMultiplier) {
    this.feedMultiplier = feedMultiplier;
    NumberOfCows = numberOfCows;
}
```

需要先设置单位饲料数，因为第二个语句调用了NumberOfCows设置存取方法，其中要求FeedMultiplier有一个值才能设置BagsOfFeed。

我们需要一种方法来区别名为feedMultiplier的字段和同名的参数。在这里"this"关键字会非常方便。

由于"this"总是当前对象的一个引用，这个this.feedMultiplier指示的是字段。如果没有"this"，feedMultiplier是指参数。所以构造函数中第一行将私有字段feedMultiplier设置为等于构造函数的第二个参数。

## there are no Dumb Questions

**问：** 可以有不带任何参数的构造函数吗？

**答：** 可以。类的构造函数没有任何参数其实相当常见。实际上，你已经见过这样的一个例子：窗体的构造函数。查看一个新增加的Windows窗体，找到它的构造函数声明：

```
public Form1() {
  InitializeComponent();
}
```

这就是窗体对象的构造函数。它没有任何参数，不过必须完成很多工作。花点时间来研究一下这个构造函数，打开Form1.Designer.cs。点击"Windows Form Designer generated code"（Windows窗体设计工具生成的代码）旁边的加号找到InitializeComponent()方法。

这个方法会初始化窗体上的所有控件，并设置这些控件的所有属性。如果在IDE的窗体设计工具中将一个新控件拖到这个窗体上，并在Properties窗口中设置它的一些属性，就会看到这些改变将在InitializeComponent()方法中反映出来。

InitializeComponent()方法在窗体构造函数内部调用，因此一旦创建窗体对象就会初始化所有控件（要记住，显示的每一个窗体也同样是对象，只不过它使用了.NET Framework在System.Windows.Forms命名空间中提供的显示窗口、按钮和其他控件的方法）。

**Watch it!**

### 如果方法参数与字段同名，则它会<u>屏蔽</u>（masks）该字段。

有没有注意到构造函数的feedMultiplier参数看起来很像FeedMultiplier属性的后备字段？如果希望在构造函数中使用这个后备字段，则必须使用"this."，feedMultiplier指示的是参数，使用this.feedMultiplier才能访问私有字段。

## there are no Dumb Questions

**问：获取或设置存取方法中为什么需要复杂的逻辑？不就是一种创建字段的方法吗？**

**答：** 因为要知道，有时每次设置一个字段时，都必须做一些计算或者完成一些动作。想想Kathleen的问题，她之所以遇到麻烦就是因为在DinnerParty类中设置了人数之后，窗体没有运行相应的方法来重新计算装饰费用。如果把这个字段替换为一个设置存取方法，就可以确保设置存取方法将重新计算装饰费用（实际上，后面几页就会让你做这个工作）！

**问：等一下，那么方法与获取或设置存取方法之间有什么区别呢？**

**答：** 没有任何区别！获取和设置存取方法只是一种特殊的方法，对其他对象来说看上去就像字段，设置该字段时就会调用。获取存取方法总是返回一个与字段类型相同的值，而设置存取方法总有一个名为value的参数，其类型与字段相同。对了，顺便说一句，可以不说“获取和设置存取方法”，而简单地称之为“属性”。

**问：那么属性中是不是可以有任何语句？**

**答：** 确实如此。方法中能做的，在属性中也都可以做。属性中可以调用其他方法、访问其他字段，甚至创建对象和实例。但是只有当访问属性时才会调用，所以如果其中的语句与获取或设置该属性无关，那就没有什么意义。

**问：既然设置存取方法总有一个名为value的参数，为什么不像其他有value参数的方法那样，在声明中加一个括号，其中包含“int value”呢？**

**答：** 这是因为C#的特别设计，它要求不必输入编译器不需要的额外信息。不必明确输入也能声明参数，如果只是要输入一个或两个参数，这可能意义不大，不过如果必须输入几百个参数，这就能大大节省时间（更何况还可以避免输入这些参数时可能出现的bug）。

每个设置存取方法总有一个名为value的参数，而且该参数的类型总是与属性类型一致。由于C#的设计，你不必再编写冗余的代码。一旦输入“set {”，C#就已经有了所需的有关类型和参数的全部信息。所以没有必要再输入更多内容，而且C#编译器也不会让你输入更多不必要的信息。

**问：请等一等，就是因为这个原因才不在构造函数中增加返回值吗？**

**答：** 完全正确!构造函数没有返回值，因为所有构造函数都肯定是void。如果在每个构造函数前面再键入“void”就是多余的，所以不必这么做。

**问：可以只有获取方法而没有设置方法吗？或者只有设置方法而没有获取方法？**

**答：** 可以！如果有一个获取存取方法而没有设置存取方法，就创建了一个只读字段。例如，SecretAgent类可能有一个只读的name字段：

```
string name = "Dash Martin";
public string Name {
    get { return name; }
}
```

如果创建一个属性，它只有设置存取方法而没有获取存取方法，那么后备字段只能写不能读。SecretAgent类可以对Password属性这样处理，这样只允许其他密探写而无法看到。

```
public string Password {
 set {
  if (value == secretCode) {
    name = "Herb Jones";
  }
}
```

实现封装时，这两种技术都很有用。

**问：我使用对象已经有一段时间了，不过还没有写过构造函数。这是不是说有些类不需要构造函数？**

**答：** 不是这样的，这只是说明，如果没有定义一个构造函数，则C#会自动建立一个无参数的构造函数。如果你定义了一个构造函数，它就不再为你自动生成这样一个构造函数了。对于封装来说这是一个很有意义的工具，因为这表明你可以选择让实例化你的类的人使用你的构造函数（但这不作为一个要求）。

**属性（获取和设置存取方法）是一种特殊类型的方法，只有在另一个类读写一个属性时才会运行。**

## Sharpen your pencil

来看这里的获取和设置存取方法。使用这个类的窗体有一个新的CableBill实例，名为thisMonth，单击按钮时将调用GetThisMonthsBill()方法。写出执行以下代码之后amountOwed变量的值。

```
class CableBill {
    private int rentalFee;
    public CableBill(int rentalFee) {
        this.rentalFee = rentalFee;
        discount = false;
    }

    private int payPerViewDiscount;
    private bool discount;
    public bool Discount {
        set {
            discount = value;
            if (discount)
            payPerViewDiscount = 2;
            else
            payPerViewDiscount = 0;
        }
    }

    public int CalculateAmount(int payPerViewMoviesOrdered) {
        return (rentalFee - payPerViewDiscount) * payPerViewMoviesOrdered;
    }
}
```

```
1. CableBill january = new CableBill(4);
   MessageBox.Show(january.CalculateAmount(7).ToString());
```

mountOwed的值是什么?

```
2. CableBill february = new CableBill(7);
   february.payPerViewDiscount = 1;
   MessageBox.Show(february.CalculateAmount(3).ToString());
```

amountOwed的值是什么?

```
3. CableBill march = new CableBill(9);
   march.Discount = true;
   MessageBox.Show(march.CalculateAmount(6).ToString());
```

amountOwed的值是什么?

## there are no Dumb Questions

**问：** **我注意到，对于某些字段你使用了大写名（首字母大写），另外一些字段的名字却使用了小写（首字母小写）。这有什么区别吗？**

**答：** 有。对你来说有区别。但是对编译器而言则没有任何区别。C#不关心你对变量如何命名，但是如果你选择了奇怪的名字，这会使你的代码可读性很差。有时如果变量名很类似，只不过一个以大写字母开头而另一个以小写字母开头，这会让人很糊涂，很容易混淆。

C#中是区分大小写的。同一个方法中可以有两个分别名为Party和party的不同变量。读起来很容易混淆，不过代码确实能顺利编译。以下是关于变量名的一些提示，可以帮助你更清楚地了解这个方面。这并不是绝对不可违反的规则，编译器不关心变量是大写还是小写，但是好的变量名可以使你的代码更可读。

1. 声明私有字段时，应当采用camelCase风格（以小写字母开头）。之所以称为camelCase，这是因为它以一个小写字母开头，后面的单词则首字母大写，所以看起来很像骆驼的驼峰。

2. 公共属性和方法采用PascalCase风格（以大写字母开头）。

3. 方法参数应当采用camelCase风格。

4. 对于某些方法，特别是构造函数，参数与字段同名。如果是这样，则参数将屏蔽字段，这说明方法中使用这个名的语句指示的是参数，而不是字段。可以使用this关键字来修正这个问题，如果在变量前增加这个关键字，就是在告诉编译器你所指的是字段，而不是参数。

## Sharpen your pencil

以下代码存在一些问题。你认为这个代码中有哪些问题，需要如何修改，请写下来。

```
class GumballMachine {
      private int gumballs;

      private int price;
      public int Price
      {
         get
         {
              return price;
         }
      }

      public GumballMachine(int gumballs, int price)
      {
              gumballs = this.gumballs;
              price = Price;
      }

      public string DispenseOneGumball(int price, int coinsInserted)
      {
              if (this.coinsInserted >= price) { // check the field
                    gumballs -= 1;
                    return "Here's your gumball";
              } else {
                    return "Please insert more coins";
              }
      }
}
```

## Sharpen your pencil Solution

写出执行以下代码之后amountOwed变量的值。

```
1. CableBill january = new CableBill(4);
   MessageBox.Show(january.CalculateAmount(7).ToString());
```

amountOwed的值是什么？

28

```
2. CableBill february = new CableBill(7);
   february.payPerViewDiscount = 1;
   MessageBox.Show(february.CalculateAmount(3).ToString());
```

amountOwed的值是什么？

won't compile

```
3. CableBill march = new CableBill(9);
   march.Discount = true;
   MessageBox.Show(march.CalculateAmount(6).ToString());
```

amountOwed的值是什么？

42

## Sharpen your pencil Solution

以下代码存在一些问题。你认为这个代码中有哪些问题，需要如何修改，请写下来。

小写price指示的是构造函数的参数，而不是字段。这行代码将参数设置为Price获取存取方法返回的值，但Price根本还没有设置过！所以它没有任何作用。如果把构造函数的参数改为首字母大写的Price，这行代码就能正确工作了。

"this"关键字放错了，没有放在适当的"gumballs"上。this.gumballs指示的是属性，而gumballs才指示参数。

这个参数屏蔽了名为price的私有字段，而注释指出这个方法要检查的是后备字段price的值。

"this"关键字放在一个不合适的参数上。本来应该放在price上，因为该字段（price）被参数price屏蔽了。

```
public GumballMachine(int gumballs, int price)
{
    gumballs = this.gumballs;
    price = Price;
}

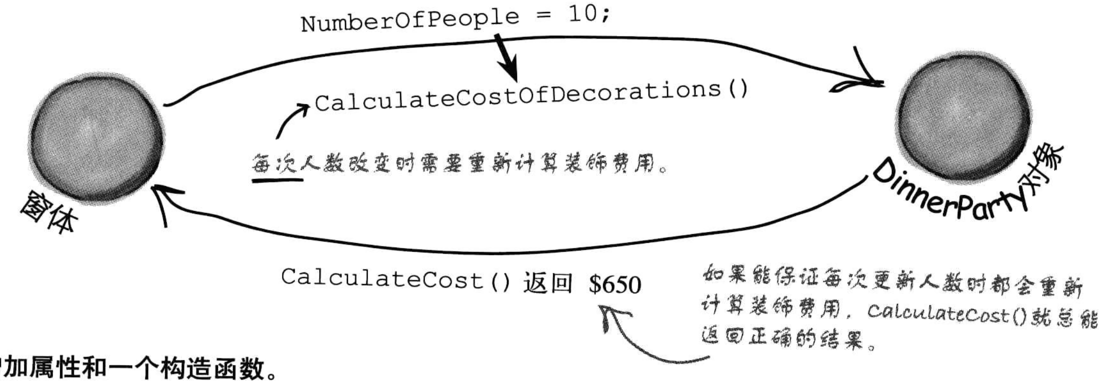
public string DispenseOneGumball(int price, int coinsInserted)
{
    if (this.coinsInserted >= price) { // check the field
        gumballs -= 1;
        return "Here's your gumball";
    } else {
        return "Please insert more coins";
    }
}
```

**Exercise**

利用前面关于属性和构造函数所学的知识，解决Kathleen聚会策划程序的问题。

**1 如何修正宴会计算程序。**

如果想修正DinnerParty类，需要确保每次NumberOfPeople改变时都会调用CalculateCostODecorations()方法。




**2 增加属性和一个构造函数。**

要解决Kathleen的问题，只需确保DinnerParty类得到适当的封装。首先将NumberOfPeople修改为一个属性，从而只要得到调用该属性就会调用CalculateCostOfDecorations()。然后增加一个构造函数，确保适当地初始化实例。最后，修改窗体来使用这个新的构造函数。如果一切正确，则对窗体做的修改仅此而已。

★ 需要为NumberOfPeople创建一个新属性，它有一个设置存取方法来调用CalculateCostOfDecorations()。它需要一个名为numberOfPeople的后备字段。

★ NumberOfPeople设置存取方法需要一个值作为参数传入CalculateCostOfDecorations()方法。所以增加一个名为fancyDecorations的私有bool字段，每次调用CalculateCostOfDecorations()时都要设置这个字段。

★ 增加一个建立类的构造函数。需要3个参数，分别对应人数（Number of People）、健康型选择（Healthy Option）以及华丽装饰选择（Fancy Decorations）。目前窗体初始化DinnerParty对象时会调用两个方法，把它们移到构造函数中。

```
dinnerParty.CalculateCostOfDecorations(fancyBox.Checked);
dinnerParty.SetHealthyOption(healthyBox.Checked);
```

★ 以下是窗体的构造函数，窗体的其他部分保持不变：

```
public Form1() {
    InitializeComponent();
    dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                     healthyBox.Checked, fancyBox.Checked);
    DisplayDinnerPartyCost();
}
```

**Exercise Solution**

利用前面关于属性和构造函数所学的知识，解决Kathleen聚会策划程序的问题。

```
class DinnerParty {
    const int CostOfFoodPerPerson = 25;

    private int numberOfPeople;
    public int NumberOfPeople {
        get { return numberOfPeople; }
        set {
            numberOfPeople = value;
            CalculateCostOfDecorations(fancyDecorations);
        }
    }
    private bool fancyDecorations;

    public decimal CostOfBeveragesPerPerson;
    public decimal CostOfDecorations = 0;

    public DinnerParty(int numberOfPeople, bool healthyOption, bool fancyDecorations) {
        NumberOfPeople = numberOfPeople;
        this.fancyDecorations = fancyDecorations;
        SetHealthyOption(healthyOption);
        CalculateCostOfDecorations(fancyDecorations);
    }

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption) {
            CostOfBeveragesPerPerson = 5.00M;
        } else {
            CostOfBeveragesPerPerson = 20.00M;
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        fancyDecorations = fancy;
        if (fancy) {
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        } else {
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
        }
    }

    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = CostOfDecorations
            + ((CostOfBeveragesPerPerson + CostOfFoodPerPerson) * NumberOfPeople);

        if (healthyOption) {
            return totalCost * .95M;
        } else {
            return totalCost;
        }
    }
}
```

现在numberOfPeople是私有的，窗体不可能改变这个字段而不重新计算装饰费用。这就能修正让Kathleen头疼的bug（要知道这个bug险些让她失去一个最好的主顾）。

通过使用属性，可以确保每次人数改变时都会重新计算装饰费用。

要注意如何使用“this.”。需要它来区别名为numberOfPeople的参数和私有字段。

所以需要在“fancyDecorations”前面加上“this.”，因为fancyDecorations参数会屏蔽同名的私有字段。

在一个字段中存储华丽装饰选择，以便NumberOfPeople设置存取方法使用这个值。

# 6 继承

# 对象的家庭树

我骑着自行车对象直冲着那个陡坡冲下去，这才想起来它是从两轮车（TwoWheeler）继承的，而我忘了给它增加一个Brakes()方法……长话短说，我缝了26针，妈妈说我得在床上呆一个月。

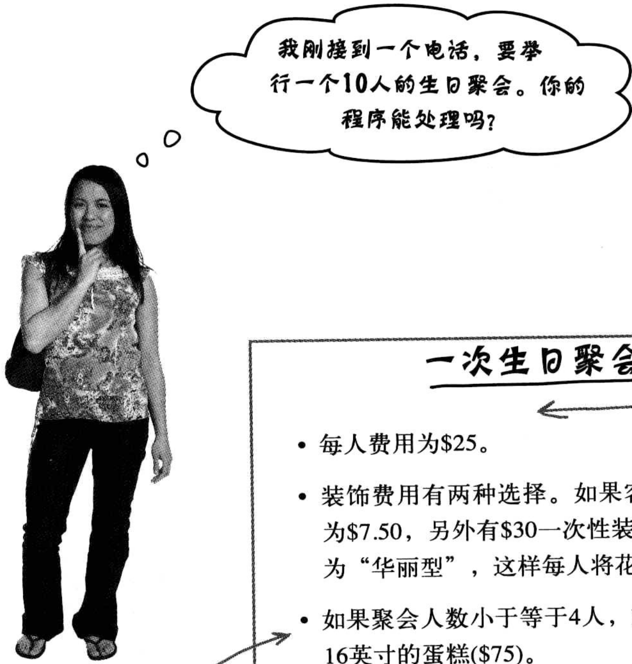



**有时你确实希望能像你的父母。**

你遇到过这样的对象吗？它几乎能完成你希望它做的所有工作，只是还差那么一点点。你是不是希望只是稍做改动这个对象就能完美无缺？我们说继承是C#语言中最为强大的概念和技术之一，以上只是其中的一个原因。读完这一章，你将了解如何派生一个对象来得到它的行为，同时还能保证灵活性，允许修改继承得到的行为。通过继承，就能避免重复代码，以更接近实际的方式对现实世界建模，最终开发的代码也将更易于维护。

# Kathleen也承办生日聚会

你的程序表现不错，所以Kathleen一直在用。不过，她不只是承办宴会，还会筹备生日聚会，报价稍有一点差异。她需要你在程序中增加策划生日聚会的功能。




这些都与宴会的要求相同。

## 一次生日聚会的花费预算

- 每人费用为$25。
- 装饰费用有两种选择。如果客户采用普通装饰，那么每人花费为$7.50，另外有$30一次性装饰费。客户也可以把聚会装饰升级为“华丽型”，这样每人将花费$15，另加$50的一次性装饰费。
- 如果聚会人数小于等于4人，就使用8英寸的蛋糕($40),否则使用16英寸的蛋糕($75)。
- 蛋糕上的文字每个字母需要$.25。8英寸的蛋糕上最多写16个字母，16英寸的蛋糕上可以最多写40个字母。

这个应用应该能够处理这两种类型的聚会。可以使用一个标签页控件，每个标签页对应一种类型的聚会。

大部分变化都与蛋糕和蛋糕上写的文字有关。

# 需要一个BirthdayParty类

要修改程序为Kathleen计算生日聚会的花费，这意味着需要增加一个新的类，并修改窗体以便处理这两种类型的聚会。

我们需要做到：

马上就会做这些事情，不过首先需要对这个任务涉及的工作有所了解。

| BirthdayParty |
|---|
| NumberOfPeople<br>CostOfDecorations<br>CakeSize<br>CakeWriting |
| CalculateCostOfDecorations()<br>CalculateCost() |

1. **创建一个新的BirthdayParty类。**
   这个新类需要计算花费、处理装饰问题，并检查蛋糕上写的文字的长度。

2. **为窗体增加一个TabControl。**
   窗体上的各个标签页非常类似赌场实验室中人们下注所用的`GroupBox`控件。只需单击要显示的标签页，并在其中拖入控件。

3. **为第一个标签页指定标签，并把处理宴会的有关控件拖入这个标签页。**
   要把处理宴会的各个控件拖到这个新标签页中。它们的工作还与从前一样，只不过只有选择了宴会标签页时才会显示。

4. **为第二个标签页指定标签，并在其中增加新的生日聚会相关控件。**
   设计一个界面来处理生日聚会，就像前面对宴会的处理一样。

5. **编写生日聚会类以及控件的相关代码。**
   现在只需要在窗体的字段中增加一个`BirthdayParty`引用，并为各个新控件增加相应的代码来使用这个`BirthdayParty`类的方法和属性。

## there are no Dumb Questions

**问：** **为什么不干脆创建DinnerParty的一个新实例？就像Mike那样，他在导航程序中比较3条路线时就是这样做的。**

**答：** 这是因为，如果创建DinnerParty类的另一个实例，就只能用它来策划另一个宴会。如果需要管理相同类型的两个不同数据，则确实可以使用同一个类的两个实例。但是如果需要存储不同类型的数据，则需要不同的类来完成。

**问：** **这个新类里该放哪些内容呢？**

**答：** 开始构建一个类之前，需要知道它要解决什么问题。正因如此，你需要与Kathleen谈谈，是她要使用这个程序。好在你得到了很多提示！你可以考虑类的行为（它需要做什么）和状态（它需要知道什么）来明确这个类的方法、字段和属性。

# 构建宴会应用2.0

开始建立一个新工程——我们将要为Kathleen的程序建立一个新版本，既可以处理宴会还可以处理生日聚会。首先创建一个充分封装的BirthdayParty类来完成具体的计算。

确保存放金额的字段和属性使用decimal类型。

**BirthdayParty**

NumberOfPeople
CostOfDecorations
CakeSize
CakeWriting

CalculateCostOfDecorations()
CalculateCost()

**动手做！**

1. **为程序增加新的BirthdayParty类。**

   你已经知道如何处理NumberOfPeople属性和CostOfDecorations方法——它们完全类似于DinnerParty中的相应属性和方法。这里首先创建这个新类，再增加NumberOfPeople属性和CostOfDecorations方法，然后再增加其余行为。

   ★ 增加一个公共的int字段CakeSize。建立一个私有方法，名为CalculateCakeSize()，它会根据人数将CakeSize设置为8或16。所以首先增加构造函数和NumberOfPeople设置存取方法。我们还将增加另外一组字段和一个常量。

```
using System.Windows.Forms;

class BirthdayParty {
    public const int CostOfFoodPerPerson = 25;

    public decimal CostOfDecorations = 0;
    private bool fancyDecorations;
    public int CakeSize;

    public BirthdayParty(int numberOfPeople,
                         bool fancyDecorations, string cakeWriting)
    {
        this.numberOfPeople = numberOfPeople;
        this.fancyDecorations = fancyDecorations;
        CalculateCakeSize();
        this.CakeWriting = cakeWriting;
        CalculateCostOfDecorations(fancyDecorations);
    }
```

一定要在类的最前面增加这个using语句，因为后面将调用MessageBox.Show()。

BirthdayParty对象初始化时，需要知道人数、华丽装饰选择以及蛋糕上的文字，以便调用CalculateCost()时能得出正确的蛋糕花费。

构造函数调用设置存取方法来设置蛋糕上的文字，为防止这个参数对于蛋糕来说过长，所以要先计算蛋糕尺寸。

构造函数设置属性，然后完成计算。

★ 需要一个CakeWriting串属性保存蛋糕上写的文字。我们将提供这个属性的相应代码。CakeWriting设置存取方法要检查CakeSize，因为不同大小的蛋糕上允许写的字母个数不同。然后使用value.Length来检查这个串有多长。如果过长，则设置存取方法不会设置私有字段，而会弹出一个消息框，指出“对一个16英寸（或8英寸）的蛋糕来说字母过多”。

★ 另外还需要CalculateCakeSize()方法。这个方法如下：

```
private void CalculateCakeSize() {
    if (NumberOfPeople <= 4)
        CakeSize = 8;
    else
        CakeSize = 16;
}
```

CalculateCakeSize()方法设置CakeSize字段。NumberOfPeople设置存取方法和CalculateCost()方法会调用这个方法。

这个属性比之前看到的属性要稍微复杂一些。它要检查蛋糕的尺寸，查看文字对于蛋糕来说是否过长。这里使用maxLength变量来存储最大长度。如果确实过长，会给出一个错误消息，然后将后备字段截断为合适的大小，从而能够重新加载到文本框中。

```
private string cakeWriting = "";
public string CakeWriting {
    get { return this.cakeWriting; }
    set {
        int maxLength;
        if (CakeSize == 8)
            maxLength = 16;
        else
            maxLength = 40;
        if (value.Length > maxLength) {
            MessageBox.Show("Too many letters for a " + CakeSize + " inch cake");
            if (maxLength > this.cakeWriting.Length)
                maxLength = this.cakeWriting.Length;
            this.cakeWriting = cakeWriting.Substring(0, maxLength);
        }
        else
            this.cakeWriting = value;
    }
}
```

CakeWriting属性确保蛋糕上的文字相对于蛋糕尺寸来说不会太长。其设置存取方法检查蛋糕尺寸，然后使用后备字段的Length属性来确保它不会过长。如果确实过长，则把这个串缩减为合适的大小。

注意到了吗？我们省略了一些大括号。如果一个代码块中只有一条语句，就不需要在两边加大括号。

每个串都有一个Substring()方法，用于返回串的一部分。这个方法可以把串截断为允许的长度，所以文本大小或蛋糕尺寸有变化时需要将文字重新加载到文本框中。

## 对于只有一行代码的代码块，大括号是可选的。

很多情况下，可能if语句或while循环的代码块中只有一行语句。如果这种情况很多，最后就会导致有大量大括号，让人看起来很不舒服！C#可以帮助你避免这个问题，如果只有一个语句，允许去掉大括号。所以以下循环和if语句的语法是完全合法的：

```
for (int i = 0; i < 10; i++)
    DoTheJob(i);
```

```
if (myValue == 36)
    myValue *= 5;
```

## 继续完成BirthdayParty类……

★ 增加CalculateCost()方法来完成BirthdayParty类。不过不是取装饰费用再加上饮料酒水的花费(DinnerParty中是这么做的)，这里会增加蛋糕的费用。

这里使用decimal，因为我们要处理价格和金额。

```
public decimal CalculateCost() {
    decimal TotalCost = CostOfDecorations + (CostOfFoodPerPerson * NumberOfPeople);
    decimal CakeCost;
    if (CakeSize == 8)
        CakeCost = 40M + CakeWriting.Length * .25M;
    else
        CakeCost = 75M + CakeWriting.Length * .25M;
    return TotalCost + CakeCost;
}
```

CalculateCost()方法与DinnerParty中的相应方法很相似，不过它会增加蛋糕的花费而不是健康型选择的相应花费。

```
private int numberOfPeople;
public int NumberOfPeople {
    get { return numberOfPeople; }
    set {
        numberOfPeople = value;
        CalculateCostOfDecorations(fancyDecorations);
        CalculateCakeSize();
        this.CakeWriting = cakeWriting;
    }
}
```

让CakeWriting属性缩减蛋糕的尺寸，这只是问题的一部分。另一部分是确保每次人数改变时都会调用CakeWriting设置存取方法。

所以当人数改变时，这个类首先重新计算蛋糕的尺寸，然后使用其CakeWriting设置存取方法来截断文字，所以如果一个10人的聚会变成4人参加，原先蛋糕上可能写有36个字母，现在就要减少来适应缩小的蛋糕。

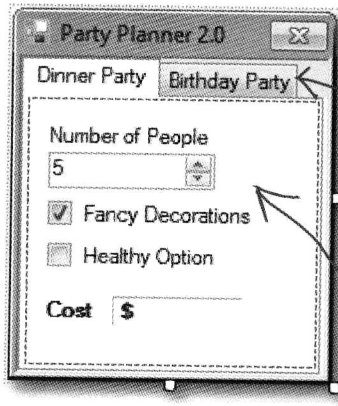
这个方法类似于DinnerParty类中的相应方法。

```
public void CalculateCostOfDecorations(bool fancy)  {
    fancyDecorations = fancy;
    if (fancy)
        CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
    else
        CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
}
}
```

## ❷ 使用TabControl为窗体增加标签页。

将一个`TabControl`从工具箱拖到窗体上，调整它的大小，使之占据整个窗体。使用`TabPages`属性修改各标签页的文本：属性窗口中这个属性旁有一个“…”按钮。单击这个按钮，IDE会弹出一个窗口，允许你编辑各个标签页的属性。将标签页的Text属性设置为“Dinner Party”和“Birthday Party”。

单击标签页实现切换。使用TabCollection属性来修改各个标签页的文本。可以单击这个属性旁边的“…”按钮，再选择各标签页的Text属性。




处理宴会的有关控件拖入这个标签页之后，只有当选择了Dinner Party标签页时它们才可见。

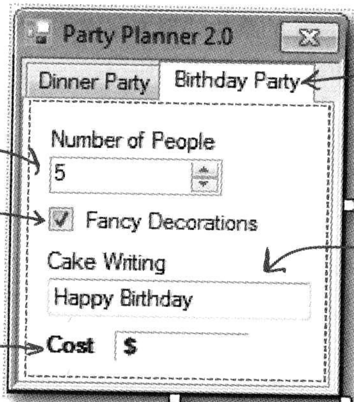
## ❸ 将宴会相关控件粘贴到“Dinner Party”标签页上。

在另一个IDE窗口中打开第5章的Party Planner程序。选择标签页上的控件，复制并粘贴到这个新Dinner Party标签页上。需要先单击这个标签页，确保将控件粘贴到正确的位置（否则会得到一个错误，指出无法将一个组件增加到类型为`TabControl`的容器中）。

这里要记住一点：将一个控件复制粘贴到一个窗体时，只是增加了控件本身，并没有增加这个控件的事件处理程序。而且需要检查Properties窗口中各个控件的（Name）设置是否正确。要确保每个控件都与第5章工程中的相应控件同名。增加了控件之后，双击各个控件，为它增加一个新的空事件处理程序。

## ❹ 建立生日聚会用户界面。

Birthday Party GUI中对应人数有一个`NumericUpDown`控件，对应华丽装饰选择有一个`CheckBox`控件，对应花费有一个带3D边框的`Label`控件。然后为蛋糕上写的文字增加一个`TextBox`控件。




单击Birthday Party标签页，并增加新控件。

这个标签页与Dinner Party标签页一样使用了NumericUpDown、CheckBox和Label控件。将它们分别命名为numberBirthday、fancyBirthday和birthdayCost。

为蛋糕上写的文字增加一个TextBox控件，名为cakeWriting（另外在这个控件上方增加一个标签，以便用户知道这个文本框做什么用）。使用文本框的Text属性指定其默认值为“Happy Birthday”。

**继续完成窗体的代码……**

**5** **集成。**

万事俱备了，现在只需要写一点代码让这些控件起作用。

- 窗体中需要有字段保存BirthdayParty对象和DinnerParty对象的引用，另外需要在构造函数中实例化这些对象。
- 你已经有宴会相关控件事件处理程序的代码（可以在第5章找到）。在Dinner Party标签页中双击NumericUpDown和CheckBox控件来增加事件处理程序。然后从第5章的程序将各个事件处理程序的内容复制粘贴到这里。窗体的代码如下：

与DinnerParty的实例一样，BirthdayParty实例在窗体的构造函数中初始化。

```
public partial class Form1 : Form {
    DinnerParty dinnerParty;
    BirthdayParty birthdayParty;
    public Form1() {
        InitializeComponent();
        dinnerParty = new DinnerParty((int)numericUpDown1.Value,
                              healthyBox.Checked, fancyBox.Checked);
        DisplayDinnerPartyCost();

        birthdayParty = new BirthdayParty((int)numberBirthday.Value,
                           fancyBirthday.Checked, cakeWriting.Text);
        DisplayBirthdayPartyCost();
    }

    // fancyBox, healthyBox和mumericUpDown1事件处理程序和
    // DisplayDinnerCost()方法与
    // 第5章最后Dinner Party练习中的相应方法完全相同。
```

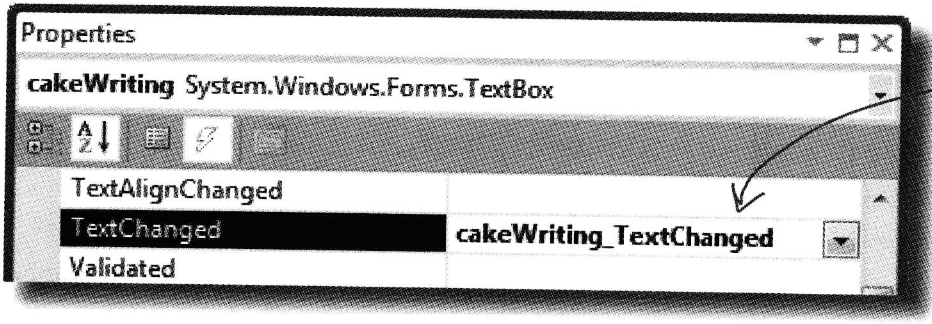
- 为NumericUpDown控件的事件处理方法增加代码，设置对象的NumberOfPeople属性，并使Fancy Decorations复选框起作用。

```
private void numberBirthday_ValueChanged(object sender, EventArgs e) {
    birthdayParty.NumberOfPeople = (int)numberBirthday.Value;
    DisplayBirthdayPartyCost();
}

private void fancyBirthday_CheckedChanged(object sender, EventArgs e) {
    birthdayParty.CalculateCostOfDecorations(fancyBirthday.Checked);
    DisplayBirthdayPartyCost();
}
```

CheckBox和NumericUpDown控件的事件处理程序与宴会相应控件的事件处理程序相同。

★ 使用Properties窗口中的Events页，为cakeWriting TextBox增加一个新的TextChanged事件处理程序。单击Properties窗口上的闪电按钮，切换到Events页。然后选择TextBox，滚动直到找到TextChanged事件。双击这个事件，为它增加一个新的事件处理程序。




选择cakeWriting TextBox并双击Properties窗口Events页中的TextChanged行时，IDE会增加一个新的事件处理程序，每次文本框中的文本变化时都会触发这个事件处理程序。

```
private void cakeWriting_TextChanged(object sender, EventArgs e) {
    birthdayParty.CakeWriting = cakeWriting.Text;
    DisplayBirthdayPartyCost();
}
```

★ 增加一个DisplayBirthdayPartyCost()方法，并把它增加到所有事件处理方法中，从而只要发生变化就会自动更新花费标签。

```
  private void DisplayBirthdayPartyCost() {
      cakeWriting.Text = birthdayParty.CakeWriting;
      decimal cost = birthdayParty.CalculateCost();
      birthdayCost.Text = cost.ToString( "c" );
  }
}
```

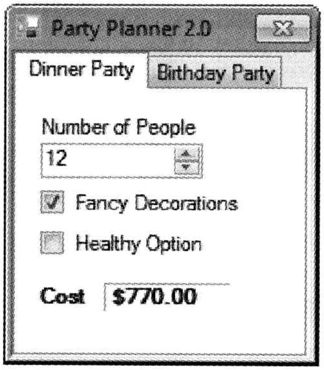
窗体处理蛋糕文字的做法相当简单，因为BirthdayParty类得到了很好的封装。窗体要做的只是使用控件来设置对象的属性，然后对象会自行负责余下的所有工作。

处理蛋糕文字、人数和蛋糕尺寸的有关逻辑都内置在NumberOfPeople和CakeWriting设置存取方法中，所以窗体只需设置并显示这些值。

……**窗体大功告成!**

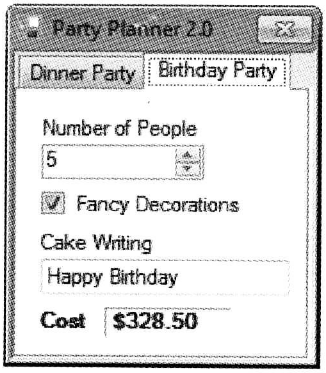
**6** **程序已经完成，现在来运行。**

确保程序能按预期的方式工作。如果文字对于蛋糕来说过长，则检查是否会弹出一个消息框。另外确保报价总是正确的。如果程序能正常工作，你的任务就完成了！




启动程序，选择Dinner Party标签页。确保它与原来的Party Planner程序工作相同。

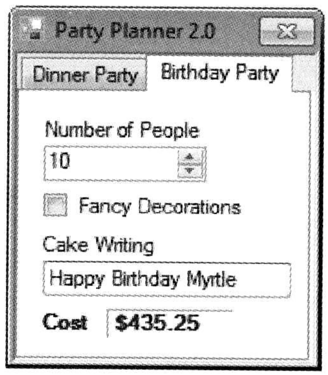
单击Birthday Party标签页。确保改变人数或单击Fancy Decorations复选框时聚会费用也会改变。




计算正确吗？在这里10个人意味着每人$25（共$250），再加上16寸蛋糕的费用（$75），另外由于选择非华丽型装饰，还要加上每人$7.50的装饰费用（共$75），以及$30的一次性装饰费，还有蛋糕上每个字母需要$.25，共21个字母，所以还要加上$5.25。

因此$250 + $75 + $75 + $30 + $5.25 = $435.25。完全正确！

在Cake Writing文本框中输入时，每次增加或减少一个字母时TextChanged事件处理程序都会更新花费。



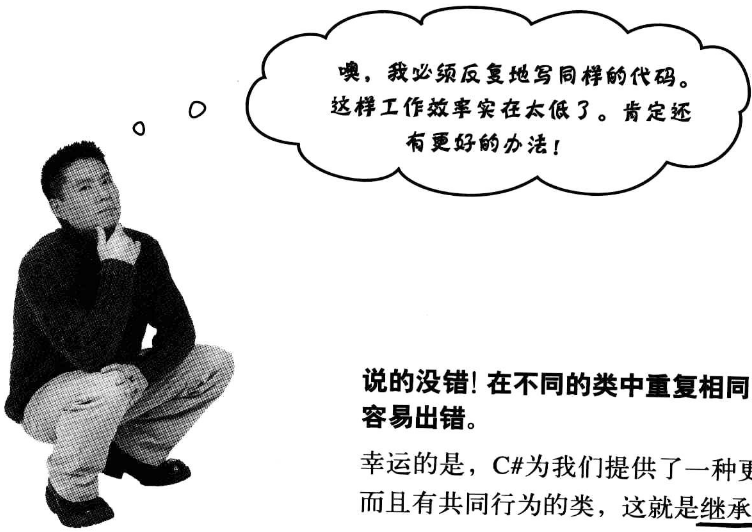
# 还有一个问题…… 如果人数超过12，聚会能额外收取$100吗？

因为你的程序让Kathleen如虎添翼，现在她拿到很多业务，由于业务太多，所以可以对一些大客户多收取一些费用。要修改程序来增加额外收取的费用，该怎么做呢？

- ★ 修改`DinnerParty.CalculateCost()`方法，检查`NumberOfPeople`，如果大于12就将返回值增加$100。
- ★ 对`BirthdayParty.CalculateCost()`做同样的修改。

花点时间来考虑这样一个问题，怎样才能为`DinnerParty`和`BirthdayParty`类都增加这个费用。要写哪些代码？应该把这些代码放在哪里？

很容易…… 不过如果有3个类似的类又该怎样？如果有4个呢？或者12个呢？如果要求你维护这个代码，并在以后做更多修改，该怎么做？如果必须对5～6个紧密相关的类做完全相同的修改，又该怎么办？

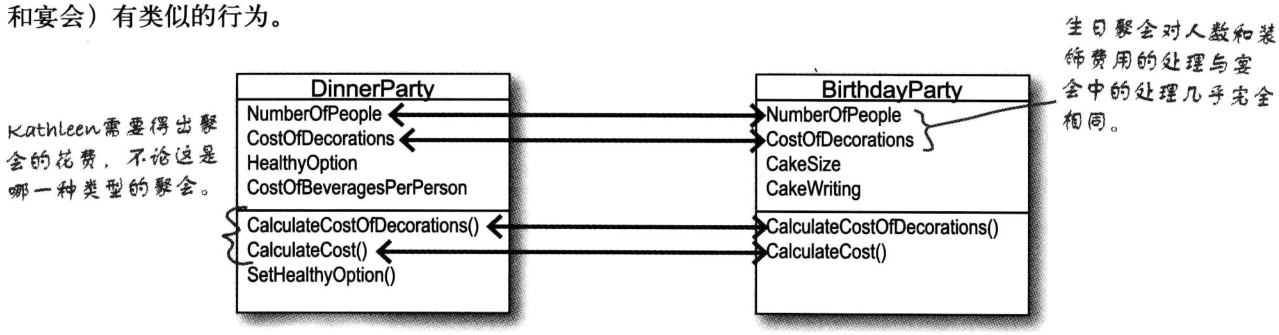



**说的没错！在不同的类中重复相同的代码确实效率低下，而且容易出错。**

幸运的是，C#为我们提供了一种更好的办法来构建彼此关联而且有共同行为的类，这就是继承。

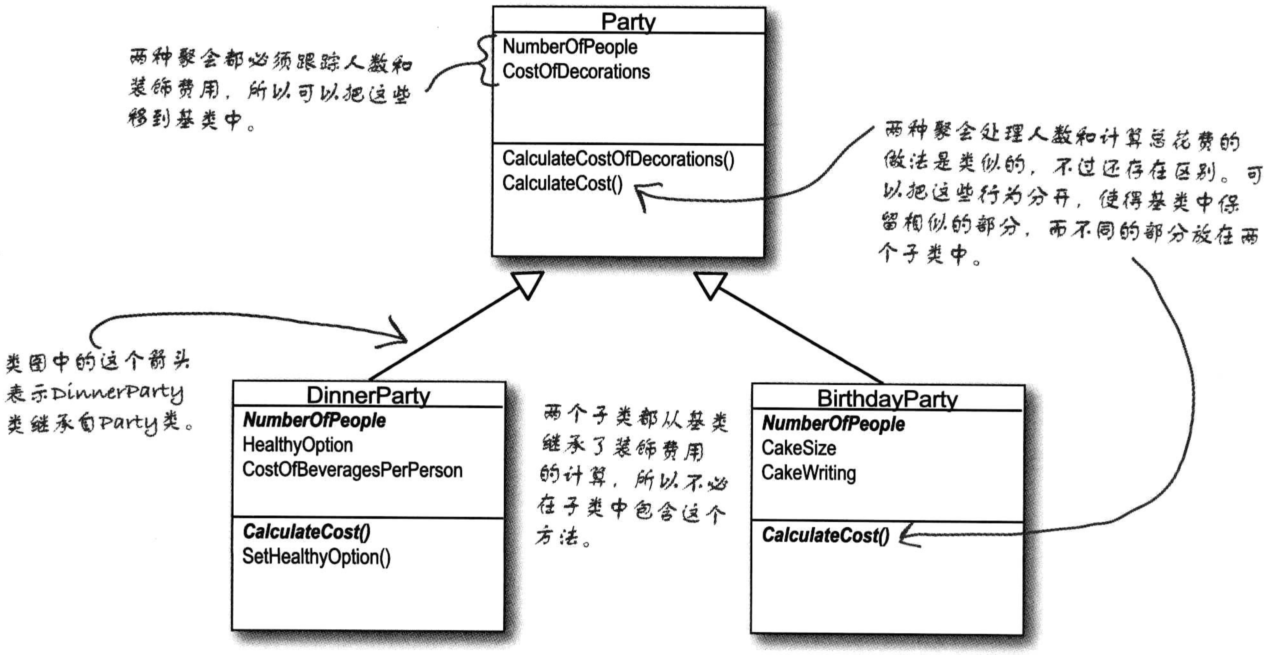
# 类使用继承时，只需写一次代码

DinnerParty和BirthdayParty类有很多相同的代码，这可不是巧合。编写C#程序时，通常会创建一些类来表示真实世界中的事物，这些事物相互之间往往存在关联。类中之所以存在类似的代码，正是因为它们表示的真实事物（生日聚会和宴会）有类似的行为。



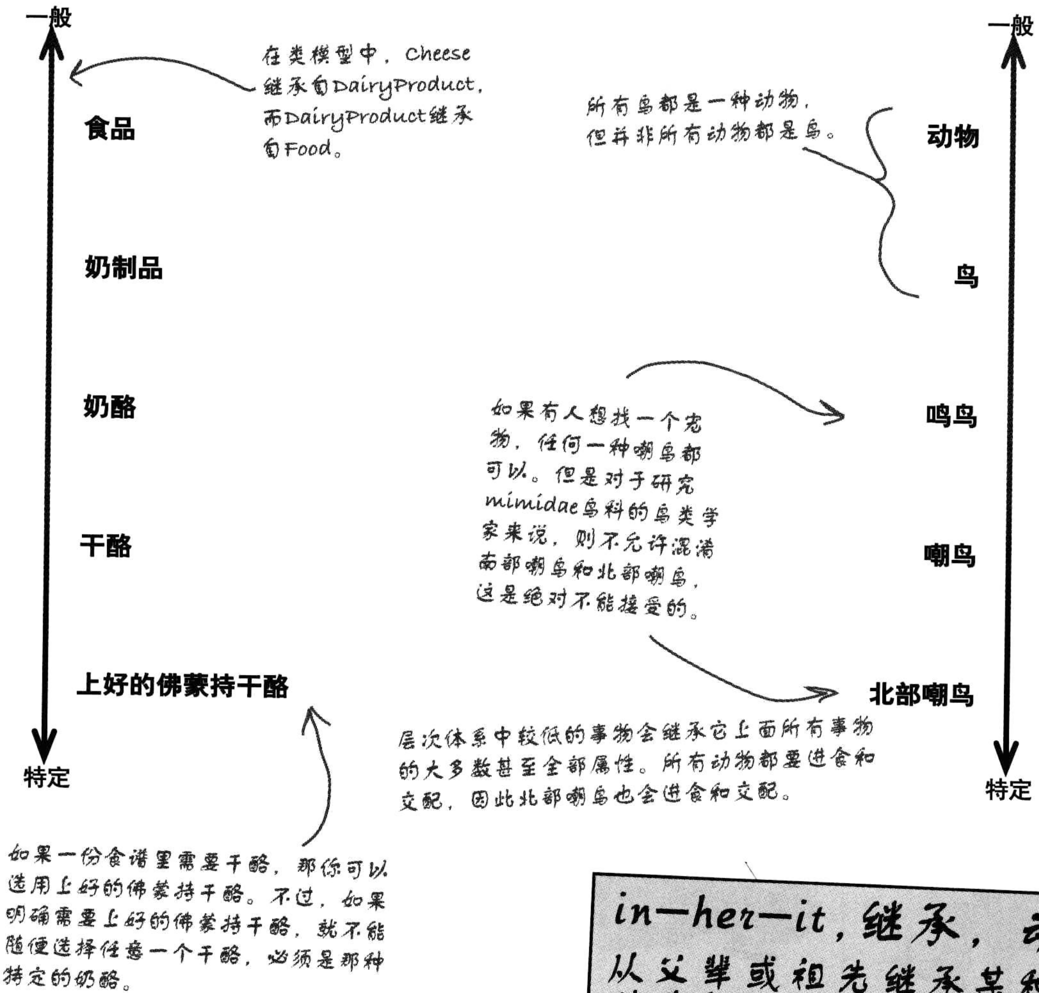

## 宴会和生日聚会都是聚会

如果两个类是某种更一般情况的特定特例，就可以把它们创建为继承(inherit）同一个类。这样一来，这两个类都是同一个基类（base)的子类（subclass）



# 建立类模型：从一般到特定

C#程序使用了继承，因为程序会对真实世界中的实际事物建模，而继承可以模仿实际事物之间的关系。真实世界中的事物往往存在于一个层次体系（hierarchy）中，从一般到特定，相应的，程序中的类层次体系（class hierarchy）也是如此。在类模型中，层次体系结构中较低的类继承自它上面的类。




**in-her-it，继承，动词。**
从父辈或祖先继承某种特性。她希望孩子能继承她棕色的大眼睛，而不要继承她丈夫蓝色的小眼睛。

# 怎样设计一个动物园模拟系统？

狮子、老虎、黑熊……噢，天哪！还有河马、狼和稀有的猫。你的任务是设计一个程序来模拟一个动物园（不要太兴奋，我们并不是具体构建代码，只是要设计一些类来表示这些动物）。

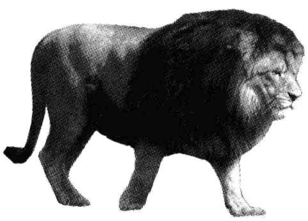
我们已经有了一个名单，其中列出了程序中将出现的一些动物，但这并不是全部。我们知道，每个动物都由一个对象表示，而这些对象会在模拟系统中“活动”，按设计完成各个动物应该做的事情。

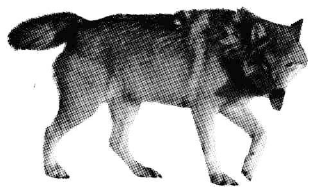
更重要的是，我们希望其他程序员能很容易地维护这个程序，这表示，以后他们可能需要增加自己的类，向这个模拟系统增加新的动物。

那么第一步要做什么？讨论特定的动物之前，需要先确定它们共同的一般特点，抽取出所有动物都有的抽象特性。然后可以把这些特性增加到一个类中，所有动物类都从这个类继承。

❶ **找出动物共同的特点。**

下面来看这6种动物。狮子、河马、老虎、猫、狼和斑点狗有什么共同的地方？它们之间有什么关系？需要找出它们的关系，这样才能得到涵盖所有这些动物的类模型。













# 使用继承避免子类中出现重复代码

关于重复代码的弊端你应该已经很清楚了。重复代码很难维护，而且总是让人很头疼。所以下面为Animal基类选择适当的字段和方法（只需写一次），以便各个动物子类都能继承。首先来看公共字段：

★ `Picture`: 放在一个PictureBox中的图片。

★ `Food`: 这种动物所吃食物的类型。目前只有两个值：肉或草。

★ `Hunger`: 这是一个int值，表示动物的饥饿程度。取决于动物何时吃（以及吃多少），这个值会有所改变。

★ `Boundaries`: 这是一个类引用，动物会在一个笼子里活动，这个类就存储了笼子的高度、宽度和位置。

★ `Location`: 动物所在位置的X和Y坐标。

Animal类有4个方法可以由动物继承：

★ `MakeNoise()`: 这个方法允许动物发出声音。

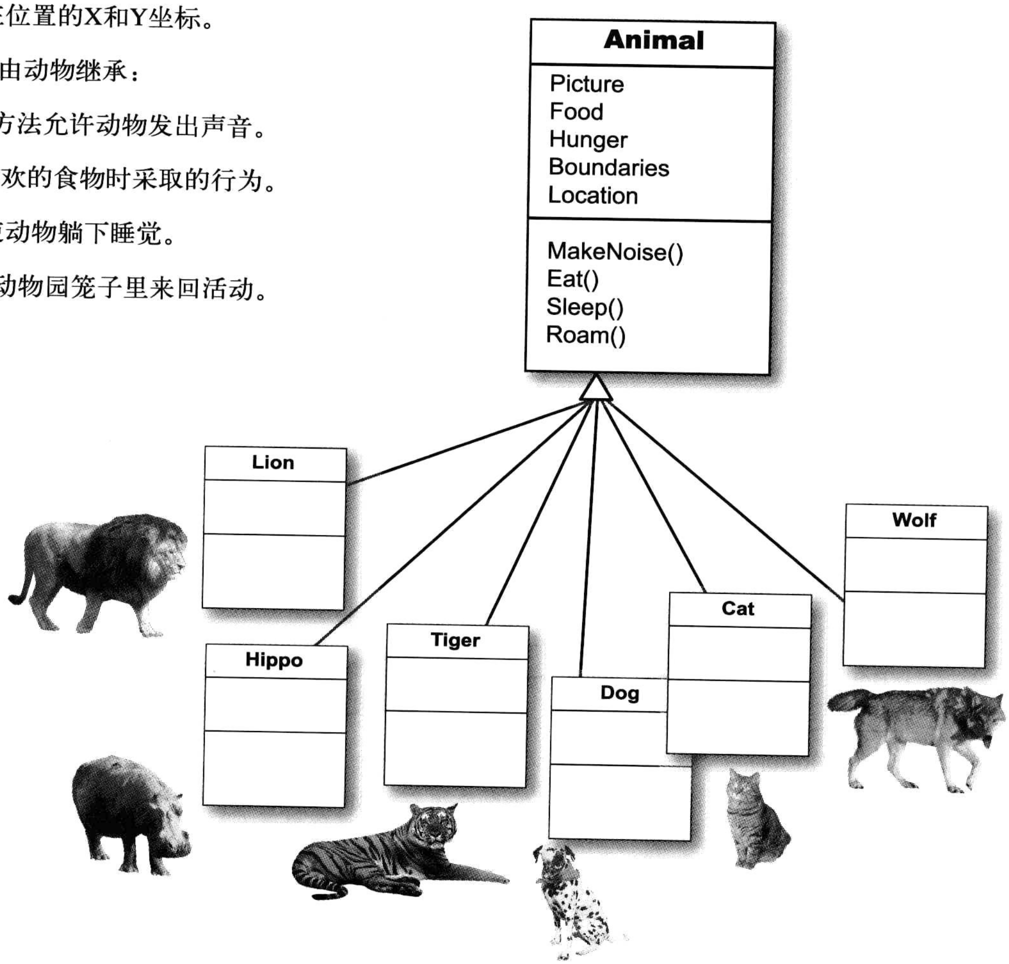
★ `Eat()`: 动物遇到它喜欢的食物时采取的行为。

★ `Sleep()`: 这个方法使动物躺下睡觉。

★ `Roam()`: 动物喜欢在动物园笼子里来回活动。

**2 构建一个基类，提供动物共有的全部特点。**

基类中的字段、属性和方法会为继承自这个基类的所有动物提供共同的状态和行为。它们都是动物，所以将基类命名为Animal很合理。




选择基类就是在做出选择。可以使用一个ZooOccupant类定义食物和养护花销，或者可以使用一个Attraction类关于动物如何取悦动物园游客提供一些方法。不过，我们认为这里还是Animal最合理，你觉得呢？

# 不同的动物会发出不同的声音

狮子会咆哮、狗会吠叫，另外就我们所知，河马根本不会发出声音。继承自Animal的各个类都会有一个MakeNoise()方法，但是每个方法的做法不同，相应地就有不同的代码。子类改变所继承的某个方法的行为时，我们称为它覆盖（overrides）了这个方法。

## 想想哪些需要覆盖

一个子类改变了所继承的某个方法的行为时，我们称之为覆盖（overriding）。每个动物都需要吃东西。不过狗可能只吃一点肉就够了，而河马要吃非常多的草。那么，这种行为的相应代码会是什么样呢？狗和河马都会覆盖Eat()方法。每次调用河马的方法时，要用去一定的干草，比如说20磅的干草。另一方面，调用狗的Eat()方法时，只会使动物园的食物储备稍有减少（一罐12盎司的狗食）。

如果有一个继承自基类的子类，它必须继承基类的所有行为……不过可以在子类中修改这些行为，从而与基类中的表现不完全一样。这正是覆盖的含义。

如果Animal基类中有一个属性或方法，这并不意味着每个子类都必须以同样的方式来使用这个属性或方法……甚至基类中可能根本没有子类中的某些成员！

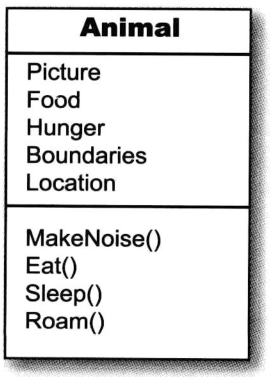
**3 找出各个动物的哪些行为与Animal类有所不同，或者哪些行为在Animal类中根本没有。**

各种动物的哪些行为是它独有的，而其他类型的动物根本没有这种行为？狗会吃狗粮，所以狗的Eat()方法需要覆盖Animal.Eat()方法。河马会游泳，所以河马有一个Swim()方法，而在Animal类中根本没有这个方法。










**BRAIN POWER**

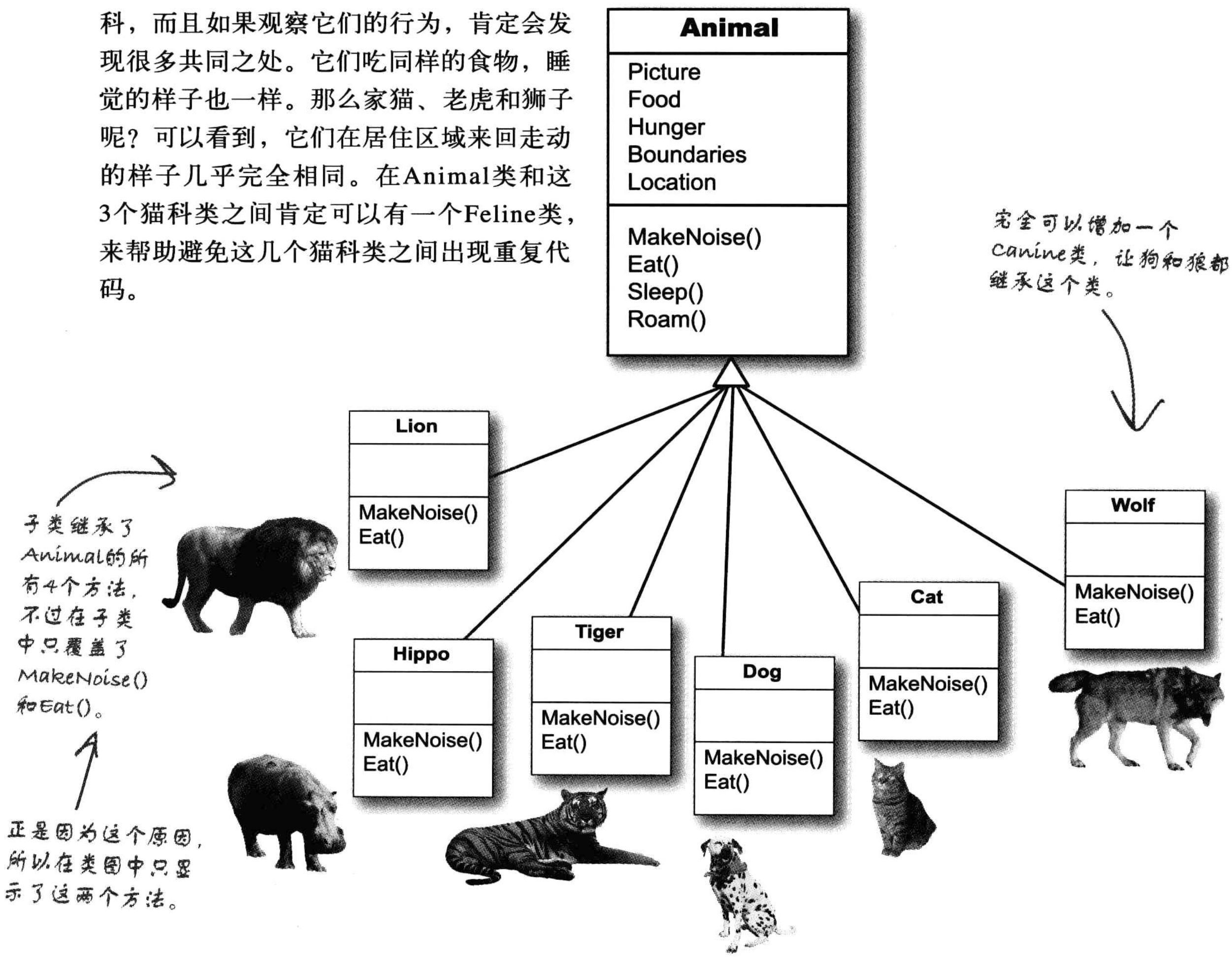
我们已经知道，有些动物会覆盖**MakeNoise()**和**Eat()**方法。哪些动物要覆盖**Sleep()**或**Roam()**呢？有没有动物会覆盖这些方法？属性呢？哪些动物会覆盖属性？

# 想想如何对动物分组

上好的佛蒙特干酪是一种奶酪，这是一种奶制品，而奶制品又是一种食品，所以一个好的食品类模型要表示出这种层次结构。很幸运，C#为我们提供了一种很容易的方法来做到这一点。可以创建类的一个链，这些类形成继承关系，从最上层的基类一直向下继承。可以有一个Food类，它有一个名为DairyProduct的子类，这个子类则作为Cheese的基类，而Cheese类又有一个子类名为Cheddar，更进一步，AgedVermontCheddar由这个Cheddar类继承。

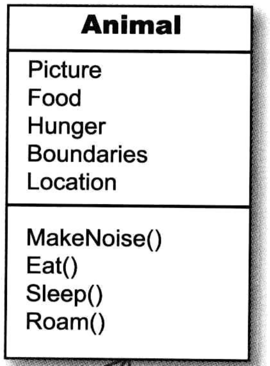
**4** **查找有共同特点的类。**

狗和狼看上去是不是很像？它们都属于犬科，而且如果观察它们的行为，肯定会发现很多共同之处。它们吃同样的食物，睡觉的样子也一样。那么家猫、老虎和狮子呢？可以看到，它们在居住区域来回走动的样子几乎完全相同。在Animal类和这3个猫科类之间肯定可以有一个Feline类，来帮助避免这几个猫科类之间出现重复代码。



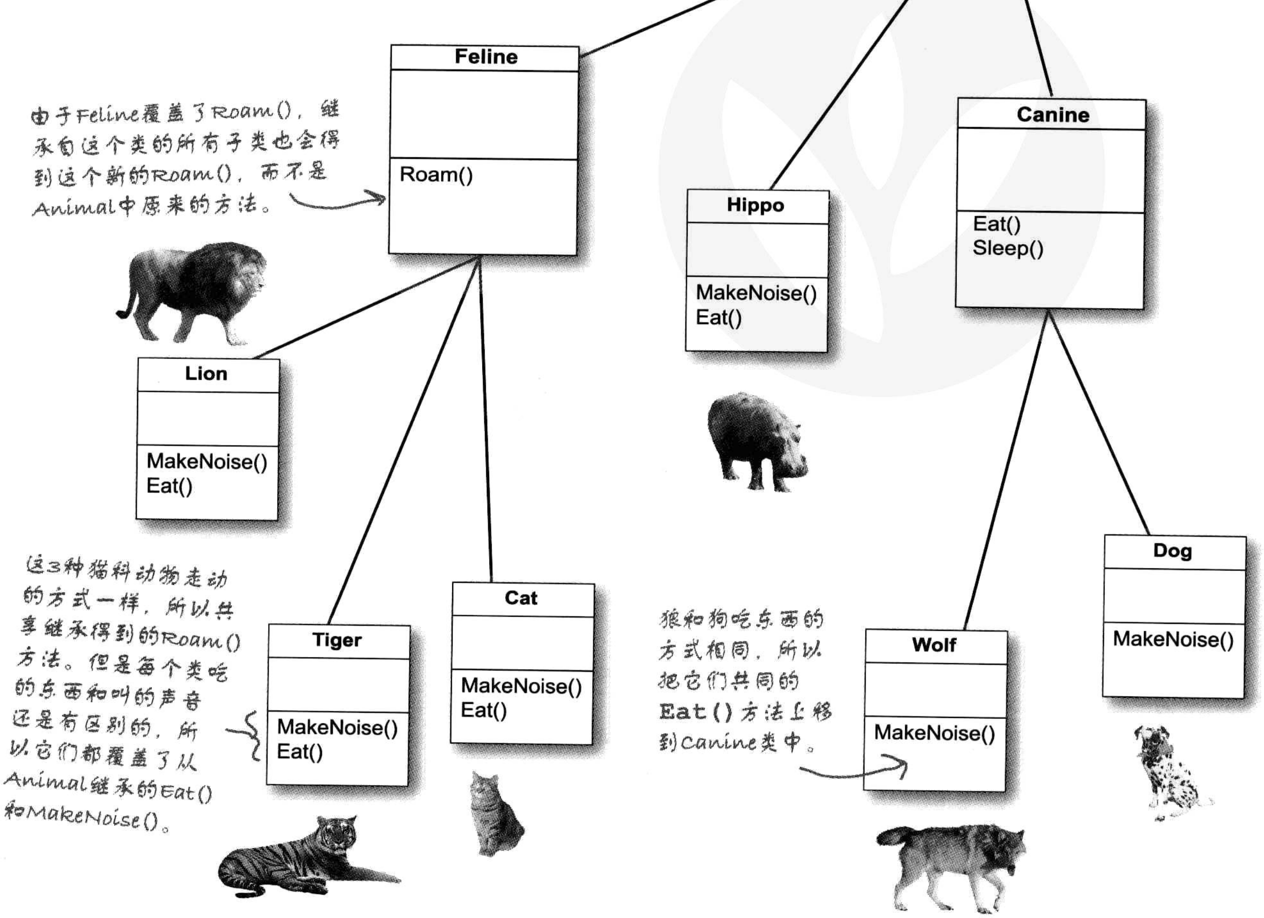
# 创建类层次体系

创建类时，最上面是一个基类，子类放在它下面，而这些子类本身又有自己的子类，此时建立的就是一个类层次体系。这就不只是能够避免重复代码了（尽管避免重复代码确实是构建合理体系结构的一个重要优点）。不过，谈到类层次体系时，最大的好处是代码将很容易理解，也很容易维护。查看这个动物园模拟系统代码时，如果看到Feline类中定义的一个方法或属性，你马上就能知道所看到的是所有猫科动物共有的行为。这个类层次体系就会成为一个导航地图，可以帮助你在程序中明确方向。




**5** **完成类层次体系。**

既然知道了如何组织这些动物，下面增加Feline和Canine类。



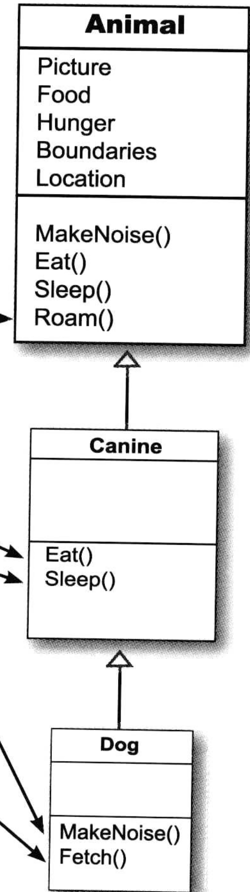
# 每个子类都扩展其基类

子类并不只是继承其基类的方法……不过你应该已经知道这一点了！毕竟，你一直在构建自己的类。为一个类增加继承关系时，所做的就是对已经构建的类进行扩展（extending），为它增加基类中的所有字段、属性和方法。所以，如果希望为狗类增加一个Fetch()方法，则这种情况相当常见。这里不存在继承或覆盖，只有狗才有这个方法，它不会出现在Wolf、Canine、Animal、Hippo或任何其他类中。

**hi-er-ar-chy,层次体系，名词。**

一种组织或分类方法，其中按上下层次标识事物或组。Dynamco总裁从一个普通的收发室工作人员做起，经过努力一直做到这个公司的高层人员。

| | |
|---|---|
| 建立一个新的Dog对象 | `Dog spot = new Dog();` |
| 调用Dog中的方法 | `spot.MakeNoise();` |
| 调用Animal中的方法 | `spot.Roam();` |
| 调用Canine中的方法 | `spot.Eat();` |
| 调用Canine中的方法 | `spot.Sleep();` |
| 调用Dog中的方法 | `spot.Fetch();` |






## C#总是调用最特定的方法

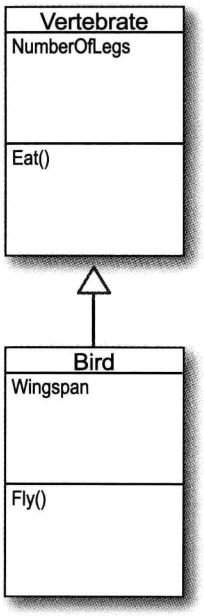
如果让狗对象四处走动，只能调用一个方法，就是Animal类中的方法。但是如果让狗发出声音呢？会调用哪一个MakeNoise()呢？

嗯，这个问题不难回答。Dog类中的MakeNoise()方法告诉你狗是如何做这件事的。如果在Canine类中，则这个MakeNoise()方法会指出所有犬科动物是怎么叫的。如果在Animal中，则这个方法就是动物发出叫声的一般描述，由于这种行为如此普遍，所以每个动物都有这种行为。因此，如果让狗发出叫声，C#首先会查看狗类，查找有没有特别针对狗的这种行为。如果Dog没有这个方法，则C#再检查Canine，然后再检查Animal。

# 使用冒号继承基类

子类从一个基类继承时，基类中所有字段、属性和方法都会自动增加到子类。

编写一个类时，可以使用一个冒号(:)，使它继承一个基类。这样这个类就成为了一个子类，并得到被继承类（基类）的所有字段、属性和方法。




```
class Vertebrate
{
    public int NumberOfLegs;
    public void Eat() {
        // code to make it eat
    }
}
```

Bird类使用一个冒号来继承Vertebrate。这说明，它会继承Vertebrate的所有字段、属性和方法。

```
class Bird : Vertebrate
{
    public double Wingspan;
    public void Fly() {
        // code to make the bird fly
    }
}
```

扩展类时，要在类声明的最后增加一个冒号，后面是所要继承的基类。

```
public button1_Click(object sender, EventArgs e) {
    Bird tweety = new Bird();
    tweety.Wingspan = 7.5;
    tweety.Fly();
    tweety.NumberOfLegs = 2;
    tweety.Eat();
}
```

tweety是Bird的一个实例，所以与以往一样，它拥有Bird的方法和字段。

由于Bird类继承自Vertebrate，Bird的每个实例都会有Vertebrate类中定义的字段和方法。

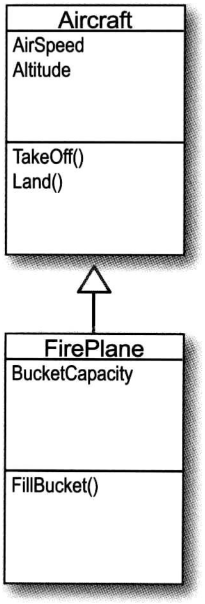
## there are no Dumb Questions

**问：** **为什么箭头向上指，从子类指向基类？如果箭头向下指，这个图不是更好看一些吗？**

**答：** 也许看上去会漂亮一些，不过这样就不正确了。建立一个类并让它继承另一个类时，会把这种关系建立在子类中，基类仍保持原样。如果从基类的角度来考虑这一点是很有道理的。增加一个从基类继承的子类时，基类的行为完全没有变化。基类甚至不知道这个新类是从它继承的。它的方法、字段和属性仍保持不变。不过，子类肯定会改变行为。子类的每个实例会自动地得到基类的所有属性、字段和方法，而所有这些只需增加一个冒号就可以轻松做到。正是因为这个原因，图中的箭头要向上指，表示这是子类的一部分，指向所继承的基类。

## Sharpen your pencil

请看以下类模型和声明，将不能正常工作的语句圈出来。

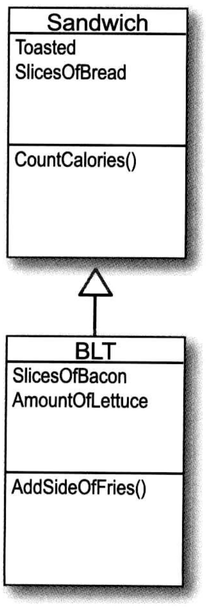



```
class Aircraft {
    public double AirSpeed;
    public double Altitude;
    public void TakeOff() { ... };
    public void Land() { ... };
}

class FirePlane : Aircraft {
    public double BucketCapacity;
    public void FillBucket() { ... };
}

public void FireFightingMission() {
    FirePlane myFirePlane = new FirePlane();
    new FirePlane.BucketCapacity = 500;
    Aircraft.Altitude = 0;
    myFirePlane.TakeOff();
    myFirePlane.AirSpeed = 192.5;
    myFirePlane.FillBucket();
    Aircraft.Land();
}
```

---



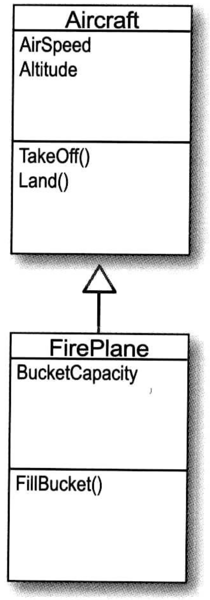

```
class Sandwich {
    public boolean Toasted;
    public int SlicesOfBread;
    public int CountCalories() { ... }
}

class BLT : Sandwich {
    public int SlicesOfBacon;
    public int AmountOfLettuce;
    public int AddSideOfFries() { ... }
}

public BLT OrderMyBLT() {
    BLT mySandwich = new BLT();
    BLT.Toasted = true;
    Sandwich.SlicesOfBread = 3;
    mySandwich.AddSideOfFries();
    mySandwich.SlicesOfBacon += 5;
    MessageBox.Show("My sandwich has "
        + mySandwich.CountCalories + "calories".);
    return mySandwich;
}
```

## Sharpen your pencil Solution

请看以下类模型和声明，将不能正常工作的语句圈出来。



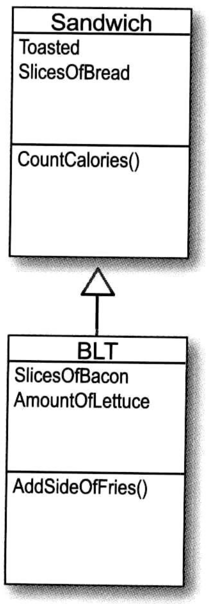

```
class Aircraft {
    public double AirSpeed;
    public double Altitude;
    public void TakeOff() { ... };
    public void Land() { ... };
}

class FirePlane : Aircraft {
    public double BucketCapacity;
    public void FillBucket() { ... };
}

public void FireFightingMission() {
    FirePlane myFirePlane = new FirePlane();
    new FirePlane.BucketCapacity = 500;
    Aircraft.Altitude = 0;
    myFirePlane.TakeOff();
    myFirePlane.AirSpeed = 192.5;
    myFirePlane.FillBucket();
    Aircraft.Land();
}
```

new关键字的用法不对。

这些语句都使用了类名而不是实例名myFirePlane。




```
class Sandwich {
    public boolean Toasted;
    public int SlicesOfBread;
    public int CountCalories() { ... }
}

class BLT : Sandwich {
    public int SlicesOfBacon;
    public int AmountOfLettuce;
    public int AddSideOfFries() { ... }
}

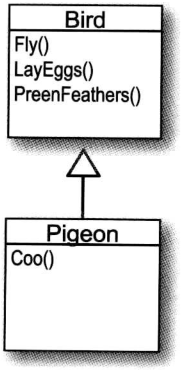
public BLT OrderMyBLT() {
    BLT mySandwich = new BLT();
    BLT.Toasted = true;
    Sandwich.SlicesOfBread = 3;
    mySandwich.AddSideOfFries();
    mySandwich.SlicesOfBacon += 5;
    MessageBox.Show("My sandwich has "
        + mySandwich.CountCalories + "calories".);
    return mySandwich;
}
```

这些属性是实例的一部分，但是这些语句却试图使用类名来调用，这是不正确的。

CountCalories是一个方法，但是这个语句在方法调用后面没有加括号()。

# 我们知道继承会为子类增加基类的字段、属性和方法……

如果子类需要继承基类的所有方法、属性和字段，则继承会很简单。




Pigeon是Bird的一个子类，所以Bird中的所有字段和方法会自动成为Pigeon的一部分。

```
class Bird {
   public void Fly() {
    // here's the code to make the bird
fly
   }

   public void LayEggs() { ... };

   public void PreenFeathers() { ... };
}

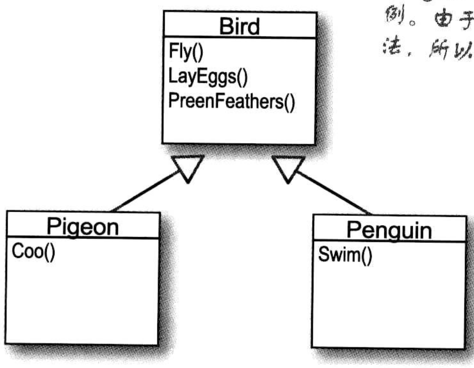

class Pigeon : Bird {
   public void Coo() { ... }
}


class Penguin : Bird {
   public void Swim() { ... }
}


public void BirdSimulator() {
      Pigeon Harriet = new Pigeon();
      Penguin Izzy = new Penguin();
      Harriet.Fly();
      Harriet.Coo();
      Izzy.Fly();
}
```

## ……但是有些鸟不会飞！

如果子类需要修改基类中的一个方法，该怎么办？




Izzy是Penguin的一个实例。由于它继承了Fly()方法，所以它完全可以飞。

Pigeon和Penguin都继承自Bird，所以它们都得到了Fly()、LayEggs()和PreenFeathers()方法。

Penguin对象不能飞！不过如果Penguin类继承自Bird，就会让企鹅也能到处飞了。这个问题该怎么解决呢？

鸽子能飞、能下蛋，还能用嘴梳理它们的羽毛，所以让Pigeon类继承Bird完全没有问题。

**BRAIN POWER**

如果这是你的Bird Simulator代码，怎么才能不让企鹅飞呢？

# 子类可以覆盖方法来改变或替换它继承的方法

有时你可能希望子类继承基类的大部分行为，但是并非所有行为。如果希望改变一个类继承的某些行为，可以覆盖（override）这些方法。

**1 为基类中的方法增加virtual关键字。**

子类只能覆盖标志有virtual关键字的方法，这就告诉C#允许子类覆盖这些方法。

```
class Bird {
   public virtual void Fly() {
      // code to make the bird fly
   }
}
```

向Fly()方法增加virtual关键字，就是在告诉C#允许子类覆盖这个方法。

**2 为派生类增加一个同名的方法。**

这个方法要有完全相同的方法签名，这说明返回值和参数都相同，而且需要在声明中使用override关键字。

为覆盖Fly()方法，向子类增加一个同名的方法，并使用override关键字。

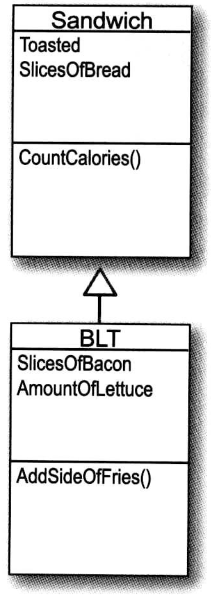
```
class Penguin : Bird {
   public override void Fly() {
     MessageBox.Show("Penguins can't fly!")
   }
}
```

覆盖一个方法时，新方法必须与基类中被覆盖的方法有相同的签名。在这里，说明方法名必须是Fly并返回void，而且没有参数。

**使用override关键字向子类增加一个方法，可以替换它继承的方法。覆盖一个方法之前，需要在基类中将这个方法标志为virtual。**

# 只要能使用基类，就可以使用它的某个子类

对于继承，最有用的一点就是可以使用子类来取代它继承的基类。所以，如果你的Recipe()方法需要一个Cheese对象，另外有一个继承自Cheese的AgedVermontCheddar类，就可以将AgedVermontCheddar的一个实例传递到Recipe()方法。不过，Recipe()只能访问属于Cheese类的字段、属性和方法，它不能访问特定于AgedVermontCheddar的任何内容。




1. 假设有一个分析三明治对象的方法：

```
public void SandwichAnalyzer(Sandwich specimen) {
    int calories = specimen.CountCalories();
    UpdateDietPlan(calories);
    PerformBreadCalculations(specimen.SlicesOfBread, specimen.Toasted);
}
```

2. 可以向这个方法传递一个三明治，不过也可以传递一个**BLT**三明治。因为**BLT**三明治是三明治的一种，建立这样一个**BLT**类，让它继承自Sandwich类。

```
public button1_Click(object sender, EventArgs e) {
    BLT myBLT = new BLT();
    SandwichAnalyzer(myBLT);
}
```

下一章还会更多地讨论这个内容。

3. 可以沿着类图下行，引用变量总是设置为等于其某个子类的实例。不过不能沿着类图上行。

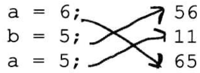
```
public button2_Click(object sender, EventArgs e) {
    Sandwich mySandwich = new Sandwich();
    BLT myBLT = new BLT();
    Sandwich someRandomSandwich = myBLT;
    BLT anotherBLT = mySandwich;     // <--- 这无法编译!!!
}
```

可以把myBLT赋给任何Sandwich变量，因为BLT是一种三明治。

但是不能将mySandwich赋给BLT变量，因为并不是所有三明治都是BLT！正因如此，最后一行代码会导致一个错误。




# 混乱的消息




以下列出了一个简短的C#程序。程序中少了一个代码块！你要解决的问题是：将候选的代码块（左边）与插入该代码块时看到的输出配对（输出在程序弹出的消息框中显示）。并不是所有输出行都会用到，另外某些输出行可能会多次用到。画线将候选代码块与其相应的输出连起来。

**要求：**

1. **填入代码中的4个空。**
2. **将候选的代码与输出配对。**

```
class A {
  public int ivar = 7;
  public ___________ string m1() {
    return "A's m1, ";
  }
  public string m2() {
    return "A's m2, ";
  }
  public ___________ string m3() {
    return "A's m3, ";
  }
}

class B : A {
  public ___________ string m1() {
    return "B's m1, ";
  }
}
```

```
class C : B {
  public ___________ string m3() {
    return "C's m3, " + (ivar + 6);
  }
}
```

这是程序的入口点——它不显示窗体，只是弹出一个消息框。

```
class Mixed5 {
  public static void Main(string[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
    string q = "";
    [          ]
    System.Windows.Forms.MessageBox.Show(q);
  }
}
```

提示：仔细想想这行代码的含义。（指向 `A a2 = new C();`）

候选代码放在这里（三行）。

**候选代码：**

```
q += b.m1();
q += c.m2();
q += a.m3();
```

```
q += c.m1();
q += c.m2();
q += c.m3();
```

```
q += a.m1();
q += b.m2();
q += c.m3();
```

```
q += a2.m1();
q += a2.m2();
q += a2.m3();
```

**输出：**

```
A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13
```

**（不要只是把以上代码键入到IDE中看结果，先在纸上动手做一做，你会学到更多东西）**



# 池塘谜题

你的任务是从这个池塘里取出一些代码片段，在代码中填空。一个代码片段可以使用多次，另外不是所有片段都要用到。你的目标是建立能够作为一个程序编译并运行的一组类。不要以为这很简单，这个问题比看上去要难。

```
class Rowboat ........................{
   public.............. rowTheBoat() {
      return "stroke natasha";
   }
}
```

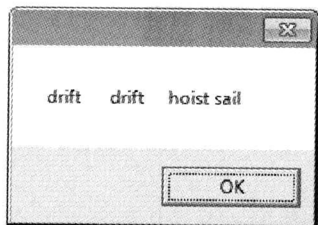
```
class ..............{
   private int...............;
  ...........void  ...............(...........) {
      length = len;
   }
   public int getLength() {
     .................;
   }
   public ..................move() {
      return   "...............";
   }
}
```

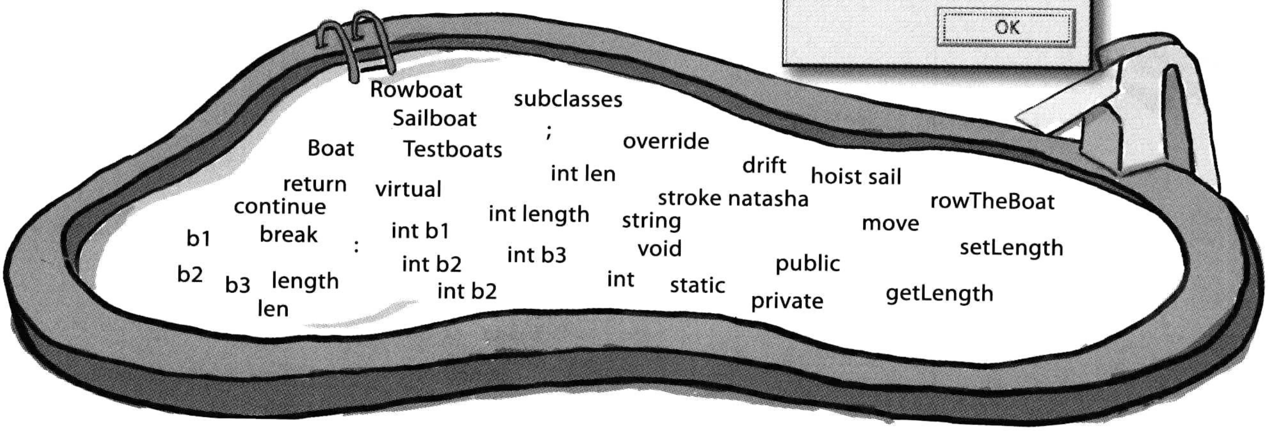
```
class TestBoats {
   ................Main(){
     ...........xyz = "";
     .........b1 = new Boat();
      Sailboat b2 = new .............();
      Rowboat .......... = new Rowboat();
      b2.setLength(32);
      xyz = b1. ..........();
      xyz += b3. ..........();
      xyz += ........move();
      System.Windows.Forms.MessageBox.Show(xyz);
   }
}
```

提示：这是这个程序的入口点。

```
class ..............: Boat {
   public ......................() {
      return "...............";
   }
}
```

**输出：**






## 混乱的消息

```
a = 6;    56
b = 5;    11
a = 5;    65
```

```
class A {
  public  virtual  string m1() {
......
  public  virtual  string m3() {
}

class B : A {
  public  override  string m1() {
......
class C : B {
  public  override  string m3() {
```

总能替换为一个子类的引用来取代基类。换句话说，总是可以使用更特定的东西来代替更一般的东西，所以如果某行代码需要一个Dog，就可以向它发送一个Canine引用。所以下面这行代码：

```
A a2 = new C();
```

表示你在实例化一个新的C对象，然后创建一个名为a2的A引用，让它指向这个C对象。A、a2和C这样的名字很适合猜谜语，不过理解起来就比较困难了。下面的几行代码也采用同样的模式，不过这些名字更容易理解：

```
Sandwich mySandwich = new BLT();

Cheese ingredient= new AgedVermontCheddar();

Songbird tweety = new NorthernMockingbird();
```

```
q += b.m1();
q += c.m2();
q += a.m3();
____________
q += c.m1();
q += c.m2();
q += c.m3();
____________
q += a.m1();
q += b.m2();
q += c.m3();
____________
q += a2.m1();
q += a2.m2();
q += a2.m3();
```

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, C's m3, 6
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```

## 池塘谜题答案



```
class Rowboat: Boat {
   public string rowTheBoat() {
      return "stroke natasha";
   }
}
class Boat {
   private int length ;
   public void setLength ( int len ) {
      length = len;
   }
   public int getLength() {
      return length ;
   }
   public virtual string move() {
      return "drift";
   }
}
```

```
class TestBoats {
   public static void Main(){
     string xyz = "";
     Boat b1 = new Boat();
     Sailboat b2 = new Sailboat();
     Rowboat b3 = new Rowboat();
     b2.setLength(32);
     xyz = b1.move();
     xyz += b3.move();
     xyz += b2.move();
     System.Windows.Forms.MessageBox.Show(xyz);
   }
}
class Sailboat : Boat {
   public override string move () {
      return " hoist sail ";
   }
}
```

## there are no Dumb Questions

**问：你在池塘谜题里提到入口点，是不是意味着程序可以没有Form1窗体？**

**答：**对。创建一个新的Windows Application工程时，IDE会为你创建这个工程的所有文件，包括Program.cs（其中包含一个带入口点的静态类）和Form1.cs（包含一个名为Form1的空窗体）。

可以做一个尝试：在IDE中创建一个新工程时，不创建新的Windows Application工程，而是创建一个空工程（选择“Empty Project”而不是“Windows Application”）。然后在Solution Explorer中为它增加一个类文件，并输入池塘谜题答案中的所有代码。由于程序使用了一个消息框，所以需要增加一个引用，为此在Solution Explorer中右键点击“References”，选择“Add Reference”，再从.NET页选择System.Windows.Forms（如果创建一个Windows Application工程，这些工作会由IDE自动为你完成）。最后，从Project菜单选择“Properties”，并选择“Windows Application”输出类型。

现在来运行这个程序……你会看到结果！祝贺你，你刚刚从头自己创建了一个C#程序。

如果需要回顾Main()方法和入口点的有关内容，则可以翻回到第2章的最前面复习！

**问：可以继承一个包含入口点的类吗？**

**答：**可以。入口点肯定是一个静态方法，但是这个方法不一定非得放在一个静态类中（要记住，static关键字表示这个类不能实例化，程序一旦运行就可以使用这个类的方法。所以在前面的池塘谜题程序中，可以从任何其他方法调用TestBoats.Main()，而不必声明一个引用变量，也不必使用new语句实例化一个对象）。

**问：我还是不太清楚为什么叫“虚”方法（virtual），对我来说，它们可是实实在在的！**

**答：**“虚”这个说法与.NET在后台如何处理这些虚方法有关。它使用了一种虚方法表（virtual method table或vtable）。.NET使用这个表来跟踪继承了哪些方法，另外哪些方法被覆盖。不用担心，使用虚方法时不要求你了解它是如何工作的！

**问：你说只能沿着类图上行而不能下行，这是什么意思？**

**答：**类图中，如果一个类在另一个类的上面，上面的这个类就比下面的类更抽象。更特定或更具体的类（如Shirt或Car）从更抽象的类（如Clothing或Vehicle）继承。如果这样来考虑就很容易看出，假如你只是要一种车，那么小汽车、货车，或者摩托车都是可以的。不过，如果需要一个小汽车，对你来说摩托车就没有什么用。

继承也是一样。如果某个方法需要一个Vehicle参数，而且Motorcycle类继承自Vehicle类，就可以将Motorcycle的一个实例传递到这个方法。但是如果方法需要Motorcycle作为参数，就不能传入任何Vehicle对象，因为这可能是一个Van实例。否则方法要访问Handlebars属性时，C#就不知道该怎么办了！

**如果一个方法需要父类作为参数，则总是可以传入其子类的实例。**



喂，我实在看不出为什么需要使用这些“virtual”和“override”关键字。如果我不使用这些关键字，则IDE只是给我一个警告，不过这个警告并没有什么影响，我的程序还是能正常运行！我的意思是说，如果这样做是“正确的”，我就会加上这些关键字，不过看来实在没有什么合适的理由让我这么做。

**使用virtual和override确实有重要的理由！**

virtual和override关键字可不是装饰。它们对于程序如何运行确实会带来很大差别。不过不要误解我的意思。下面给出一个实际例子来说明这两个关键字的作用。

**动手做！**

不要创建一个Windows Forms应用，这里要创建一个新的控制台应用！这说明这个应用没有窗体。

❶ **创建一个新的控制台应用，并增加一些类。**

在Solution Explorer中鼠标右键单击这个工程，像往常一样增加类。需要增加以下5个类：Jewels、Safe、Owner、Locksmith和JewelThief。

❷ **为这些新类增加代码。**

以下是所增加的5个新类的代码：

```
class Jewels {
    public string Sparkle() {
        return "Sparkle, sparkle!";
    }
}
```

Safe对象的contents字段中有一个Jewels引用。除非调用Open()时提供了正确的组合，否则不会返回这个引用。

注意private关键字如何隐藏contents和combination。

```
class Safe {
    private Jewels contents = new Jewels();
    private string safeCombination = "12345";
    public Jewels Open(string combination)
    {
        if (combination == safeCombination)
            return contents;
        else
            return null;
    }
    public void PickLock(Locksmith lockpicker) {
        lockpicker.WriteDownCombination(safeCombination);
    }
}
```

Locksmith可以打开组合锁，通过调用PickLock()方法并传入他自己的一个引用可以得到这个组合。Safe要根据这个组合调用其WriteDownCombination()方法。

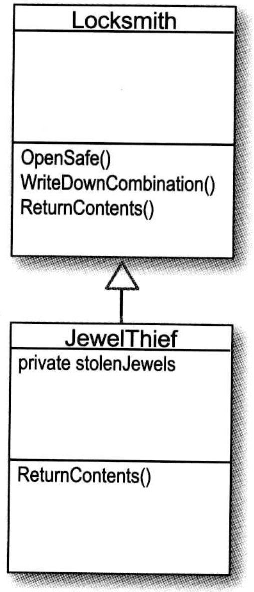
**控制台不使用窗体**

如果创建一个控制台应用而不是Windows Forms应用，则IDE为你创建的只是一个名为Program的新类，包含一个空的Main()入口点方法。运行这个应用时，它会弹出一个命令行窗口来显示输出。可以在附录A了解有关控制台的更多内容。

```
class Owner {
    private Jewels returnedContents;
    public void ReceiveContents(Jewels safeContents) {
        returnedContents = safeContents;
        Console.WriteLine("Thank you for returning my jewels! " + safeContents.Sparkle());
    }
}
```

**3** **JewelThief类继承自Locksmith。**

珠宝贼是打坏主意的锁匠！他们可以打开保险箱上的锁，不过不是把珠宝还给本来的主人，他们会把珠宝窃为己有！




```
class Locksmith {
    public void OpenSafe(Safe safe, Owner owner) {
        safe.PickLock(this);
        Jewels safeContents = safe.Open(writtenDownCombination);
        ReturnContents(safeContents, owner);
    }

    private string writtenDownCombination = null;
    public void WriteDownCombination(string combination) {
        writtenDownCombination = combination;
    }

    public void ReturnContents(Jewels safeContents, Owner owner) {
        owner.ReceiveContents(safeContents);
    }
}
```

Locksmith的OpenSafe()方法会打开锁，打开保险箱，把里面的珠宝还给主人。

```
class JewelThief : Locksmith {
    private Jewels stolenJewels = null;
    public void ReturnContents(Jewels safeContents, Owner owner) {
        stolenJewels = safeContents;
        Console.WriteLine("I'm stealing the contents! " + stolenJewels.Sparkle());
    }
}
```

JewelThief对象继承了OpenSafe()和WriteDownCombination()方法。但是OpenSafe()方法调用ReturnContents()把珠宝还给主人时，JewelThief会将其窃为自有！

**4** **这是Program类的Main()方法。**

不过先别运行！运行程序前，想想看控制台上会显示什么。

```
class Program {
    static void Main(string[] args) {
        Owner owner = new Owner();
        Safe safe = new Safe();
        JewelThief jewelThief = new JewelThief();
        jewelThief.OpenSafe(safe, owner);
        Console.ReadKey();
    }
}
```

ReadKey()等待用户按一个按键。这条语句的目的是防止程序终止。

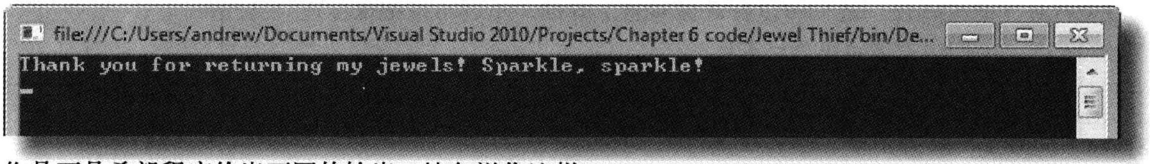
## Sharpen your pencil

仔细查看这个程序的代码。运行程序前，你认为控制台上会显示什么，请把你的想法写下来（提示：想想看JewelThief从Locksmith继承了什么）。

..............................................................................................................................

# 子类可以隐藏超类中的方法

继续运行JewelThief程序。由于这是一个控制台应用，所以并不是将其控制台输出写入Output窗口，它会弹出一个命令行窗口，输出就显示在这里。你会看到以下结果：



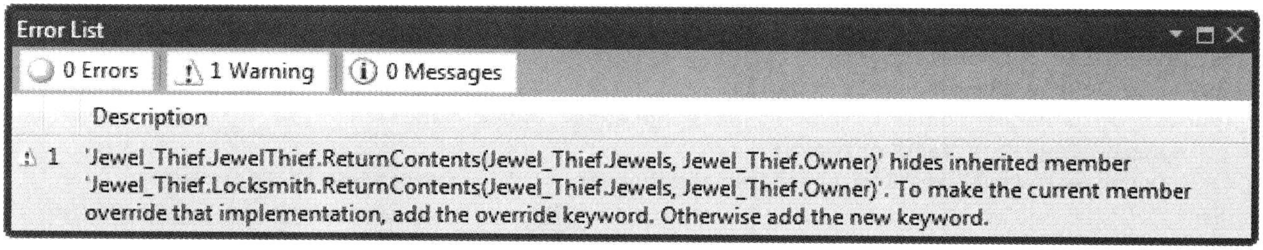

你是不是希望程序给出不同的输出？比如说像这样：

```
I'm stealing the contents! Sparkle, sparkle!
```

看起来JewelThief的做法就像是一个锁匠！怎么回事？

## 隐藏方法与覆盖方法

JewelThief对象调用ReturnContents()方法时就像一个Locksmith对象，其原因在于JewelThief类声明ReturnContents()方法的方式。编译程序时你会得到一个警告消息，从中可以发现一个很重要的提示：




由于JewelThief类继承自Locksmith，并用自己的方法替换了ReturnContents()方法，所以看起来JewelThief覆盖了Locksmith的ReturnContents()方法。但是事实上并非如此。你可能以为JewelThief会覆盖这个方法（稍后将讨论有关内容），不过实际上JewelThief只是隐藏了这个方法。

这里有一个重要的区别。子类隐藏方法时，它会替换（理论上讲，应该是“重新声明”）基类中同名的方法。所以现在我们的子类实际上有两个同名的不同方法：一个从其基类继承得来；另一个是子类中定义的全新的方法。

如果子类只是增加一个与超类方法同名的方法，那么它只是隐藏了超类方法而不是覆盖这个方法。

## 使用不同的引用来调用隐藏的方法

JewelThief只是隐藏了ReturnContents()方法（而不是扩展这个方法），这就导致在作为一个Locksmith对象调用时JewelThief就像是一个Locksmith对象。JewelThief从Locksmith继承了一个ReturnContents()版本，它自己又定义了这个方法的第二个版本，这说明，现在有两个同名的不同方法。这意味着你的类需要两种不同的方式来调用这个方法。

另外，实际上也必须如此。如果有一个JewelThief实例，则可以使用一个JewelThief引用变量来调用这个新ReturnContents()方法。不过如果使用一个Locksmith引用变量来调用这个方法，所调用的就是隐藏的Locksmith ReturnContents()方法。

```
// JewelThief子类隐藏了Locksmith基类中的一个方法，
// 所以根据你使用哪个引用来调用方法，
// 会从同一个对象得到不同的行为！

// 将JewelThief对象声明为一个Locksmith引用，这会使
// 它调用基类的ReturnContents()方法
Locksmith calledAsLocksmith = new JewelThief();
calledAsLocksmith.ReturnContents(safeContents, owner);

// 将JewelThief对象声明为一个JewelThief引用，这会使
// 它调用JewelThief的ReturnContents()方法，因为它隐藏
// 了基类的同名方法。
JewelThief calledAsJewelThief = new JewelThief();
calledAsJewelThief.ReturnContents(safeContents, owner);
```

## 隐藏方法时使用new关键字

再来仔细看这个警告消息。当然，大部分警告我们都不会一一仔细查看，对不对？不过这一次确实应该看看它是怎么说的，要让当前成员覆盖这个实现，需要增加override关键字。否则需要增加new关键字。

再回到程序中增加**new**关键字：

```
new public void ReturnContents(Jewels safeContents, Owner owner) {
```

一旦向JewelThief类的ReturnContents()方法声明增加了new，这个错误消息就会立即消失。不过你的程序还是达不到你预期的效果！它还是会调用Locksmith对象中定义的ReturnContents()方法。为什么呢？因为ReturnContents()方法是从Locksmith类定义的一个方法调用的。具体来说，就是在Locksmith.OpenSafe()方法中调用的（尽管它由一个JewelThief对象启动）。如果JewelThief只隐藏了ReturnContents()方法，则它自己的ReturnContents()就永远也得不到调用。

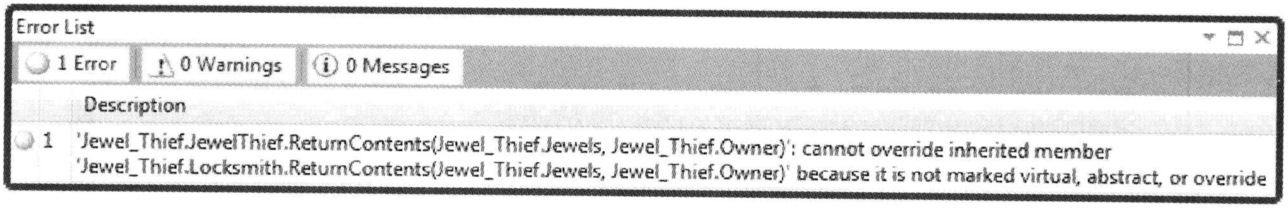
**你能想到如何让JewelThief覆盖ReturnContents()方法而不只是隐藏这个方法吗？在翻到下一页之前看看你能不能得出答案！**

# 使用override和virtual关键字来继承行为

我们真的希望JewelThief类总是使用它自己的ReturnContents()方法，而不管如何调用这个方法。这也是大多数情况下我们所期望的继承行为，这称为覆盖（overriding）。要让你的类做到这一点非常容易。首先需要在声明ReturnContents()方法时使用override关键字，如下所示：

```
class JewelThief {
    ...
    override public void ReturnContents
            (Jewels safeContents, Owner owner)
```

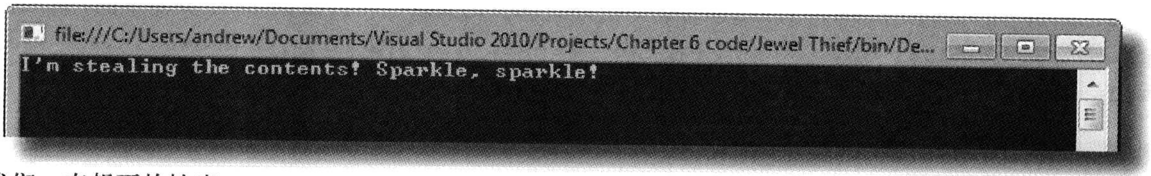
不过所要做的还不只这些。如果你只是增加了这个override就试图编译，你会得到类似下面的错误：




这一次同样需要仔细查看这个错误。JewelThief未能覆盖继承的成员ReturnContents()，因为这个方法在Locksmith中没有标志virtual、abstract或override。不过，在Locksmith可以很轻松地修正这个错误！只需要对Locksmith的ReturnContents()标志virtual关键字：

```
class Locksmith {
    ...
    virtual public void ReturnContents
            (Jewels safeContents, Owner owner)
```

现在再来运行你的程序。你会看到下面的输出：




这正是我们一直想要的输出。



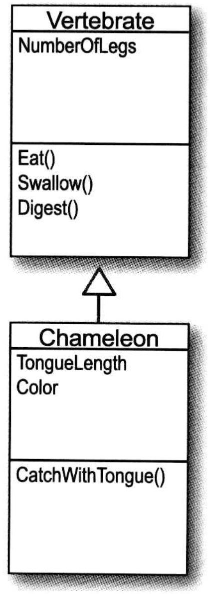

**完全正确。大多数情况下你都是希望覆盖方法，不过确实可以隐藏方法。**

处理一个扩展了一个基类的子类时，往往希望覆盖而不是使用隐藏。所以如果看到编译器警告你隐藏了一个方法，一定要注意！要确保你确实希望隐藏这个方法，而不是忘记了使用virtual和override关键字。如果你总是正确地使用virtual、override和new关键字，就再也不会遇到这种问题了！

**如果你想覆盖一个基类中的方法，则一定要标志virtual关键字，而且只要希望在子类中覆盖这个方法就要使用override关键字。如果没有做到这一点，则最终就会无意中隐藏方法。**

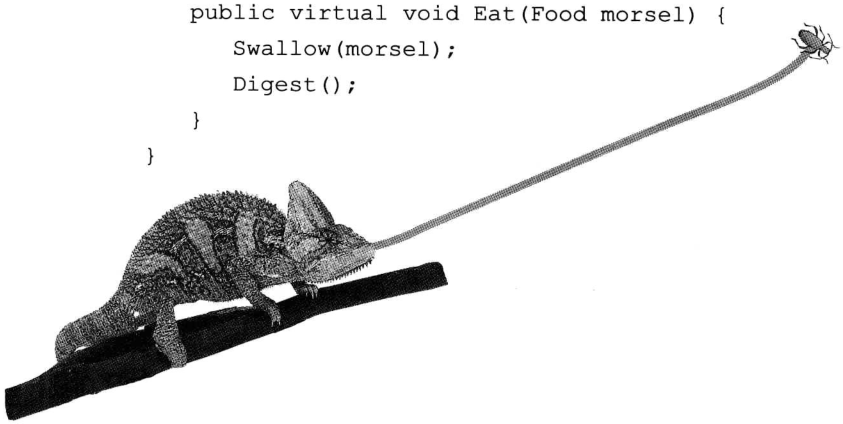
# 子类可以使用base关键字访问其基类

即使子类中覆盖了基类中的方法或属性，有时仍可能希望访问基类中的这个成员。幸运的是，可以使用**base**关键字，利用这个关键字就可以访问基类中的任何方法。




❶ 所有动物都要吃东西，所以Vertebrate类有一个Eat()方法，并取一个Food对象作为参数。

```
class Vertebrate {
   public virtual void Eat(Food morsel) {
      Swallow(morsel);
      Digest();
   }
}
```



❷ 变色龙用它们的舌头捕食。所以Chameleon类继承自Vertebrate，但是覆盖了Eat()。

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      Swallow(morsel);
      Digest();
   }
}
```

变色龙需要吞咽并消化食物，这与所有其他动物一样。不过，我们确实需要重复这些代码吗？

❸ 不必重复代码，可以使用base关键字调用所覆盖的方法。现在新版本和老版本的Eat()方法都可以访问。

```
class Chameleon : Vertebrate {
   public override void Eat(Food morsel) {
      CatchWithTongue(morsel);
      base.Eat(morsel);
   }
}
```

这行代码调用Chameleon继承的基类(Vertebrate)中的Eat()方法。

既然你已经掌握了继承的一些基本思想，下面来考虑一个问题。重用代码是减少输入的一种很好的方法，不仅如此，继承还有一个方面很有意义，这就是继承会使以后的代码维护更为容易。你能想出为什么这么说吗？

# 如果基类有构造函数，子类也需要有构造函数

如果类有一些带参数的构造函数，继承自这个类的所有子类必须调用其中某个构造函数。子类的构造函数与基类构造函数可以有不同的参数。

在子类构造函数声明的最后增加这行代码，告诉C#：每次实例化子类时都需要调用基类的构造函数。

```
class Subclass : BaseClass {
    public Subclass(parameter list)
        : base(the base class's parameter list) {
        // first the base class constructor is executed
        // then any statements here get executed
    }
}
```

这是子类的构造函数。

**基类构造函数在子类构造函数之前执行。**

不过别光听我们说，你自己试试看！

**动手做！**

1. **创建一个基类，其构造函数弹出一个消息框。**
然后为窗体增加一个按钮，实例化这个基类并显示一个消息框：

```
class MyBaseClass {
    public MyBaseClass(string baseClassNeedsThis) {
        MessageBox.Show("This is the base class: " + baseClassNeedsThis);
    }
}
```

这是基类构造函数需要的参数。

2. **尝试增加一个子类，但是不调用构造函数。**
然后向窗体增加一个按钮，实例化这个子类并显示一个消息框：

```
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue) {
        MessageBox.Show("This is the subclass: " + baseClassNeedsThis
            + " and " + anotherValue);
    }
}
```

在IDE中选择"Build""Build Solution"，会得到这个代码错误。

1 No overload for method 'MyBaseClass' takes '0' arguments

注意这个有点费解的错误。它表示子类没有调用基类构造函数。

3. **修正这个错误，让这个构造函数调用基类的构造函数。**
然后实例化子类，注意两个消息框弹出的顺序！

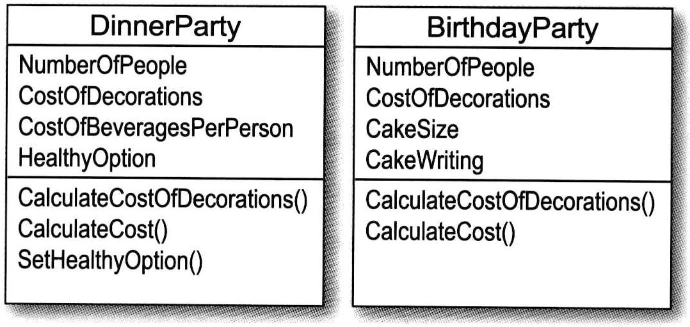
```
class MySubclass : MyBaseClass{
    public MySubclass(string baseClassNeedsThis, int anotherValue)
        : base(baseClassNeedsThis)
    {
        // the rest of the subclass is the same
```

要这样向基类构造函数传入它需要的参数。

增加这行代码，告诉C#调用基类中的构造函数。它有一个参数表，显示了向基类构造函数传递哪些参数。这样错误就没有了，可以增加一个按钮看看两个消息框如何弹出！

# 现在准备完成Kathleen布置的任务!

上一次离开Kathleen时，你已经完成了任务，为程序增加了处理生日聚会的功能。她要求如果聚会人数超过12人的话，就需要再付$100。看上去好像必须把同样的代码写两次，为每个类分别写一次。既然已经知道了如何使用继承，就可以让这两个类继承同一个基类，其中包含所有共享代码，这样一来就只需写一次代码。



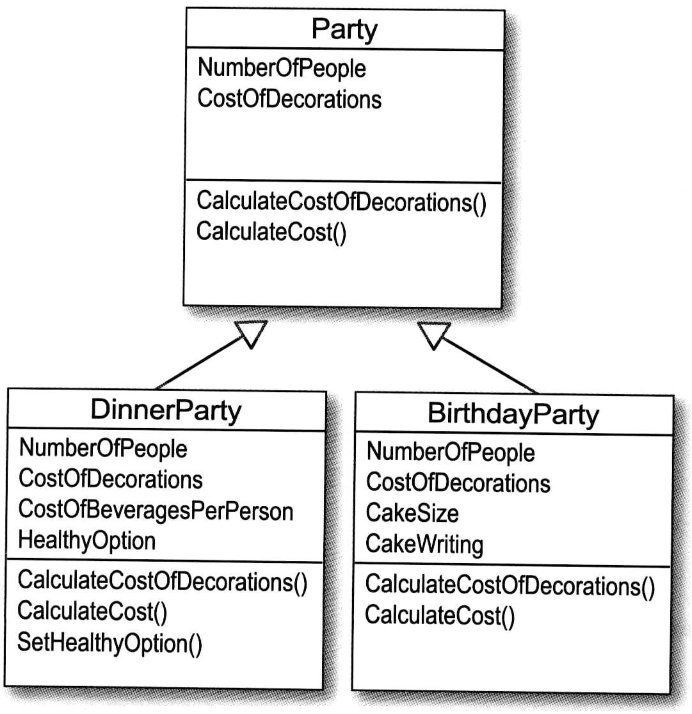




**如果方法得当，应该能修改这两个类而无需对窗体做任何改变!**

**1** **下面来创建这个新的类模型。**

仍然使用原来的`DinnerParty`和`BirthdayParty`类，但是现在它们要继承一个`Party`类。这两个类需要的方法、属性和字段基本上相同，所以不必对窗体做任何修改。不过，其中一些方法、属性和字段要移到Party基类中，另外有些方法可能必须覆盖。



**❷ 构建Party基类。**

创建Party类，确保它是公共类。需要仔细查看类图中的属性和方法。找出哪些属性和方法需要从DinnerParty和BirthdayParty中移动并放到Party中。

- ★ 将NumberOfPeople和CostOfDecorations属性移到Party中，使DinnerParty和BirthdayParty有一致的属性。

后面你将学到"protected"关键字。受保护的（protected）字段对于子类来说是公共的，对于其他类来说则是私有的。

- ★ 对CalculateCostOfDecorations()和CalculateCost()方法做同样的处理。如果这些方法需要私有字段，还需要移动相应的私有字段（要记住，子类只继承公共字段，如果将一个私有字段移入Party，DinnerParty和BirthdayParty类将不能访问这个字段）。
- ★ 还需要一个构造函数。请仔细看BirthdayParty和DinnerParty构造函数，它们共同的代码都要移到基类的构造函数中。
- ★ 现在为超过12人的聚会另外增加$100的费用。毕竟，我们现在做的所有工作就是为了增加这个功能！这是生日聚会和宴会共同的要求，所以它要作为Party的一部分。

**❸ 让DinnerParty继承Party。**

既然Party做了很多原先DinnerParty做的事情，下面可以去掉重叠的部分，只在DinnerParty中保留宴会特有的内容。

- ★ 确保构造函数能正常工作。它要完成Party构造函数不做的事情吗？如果是，保留这些内容，然后把其他内容都移到基类构造函数中。
- ★ 与设置健康型选择有关的逻辑都应当留在DinnerParty中。
- ★ 唉呀，如果想保证窗体代码不变，这里就不能覆盖CalculateCost()方法，因为我们的窗体要求传入一个名为healthyOption的bool参数。所以，我们需要重载（overload）这个方法，重载的意思就是向类增加一个有不同参数的新的CalculateCost()方法。所以仍然使用本章开始时所用的同一个方法声明。不过可以充分利用继承，通过调用base.CalculateCost()访问Party类中的CalculateCost()方法。

重载的有关内容将在第8章介绍，这里只是简单提一下，让你有所认识以便于后面的学习。

**❹ 让BirthdayParty继承Party。**

对BirthdayParty做同样的处理，所有不是特定于生日聚会的内容都移到基类中，BirthdayParty中只留下生日聚会特定的功能。

- ★ BirthdayParty构造函数中哪些工作不属于Party构造函数？
- ★ 需要在BirthdayParty中处理蛋糕的花费。这涉及一个方法和一个属性，所以需要覆盖相应的方法和属性。
- ★ 不错，确实可以覆盖属性！这与覆盖方法是一样的。设置base.NumberOfPeople的值时，它会调用基类中该属性的设置存取方法。要获取和设置这个值都需要使用base关键字。

**Exercise Solution**

试试看，你已经修改了DinnerParty和BirthdayParty 类，让它们继承同一个基类Party。接下来可以修改花费计算来增加$100的费用，而且根本不用对窗体做任何改变。真的很不错！

```
class Party
{
    const int CostOfFoodPerPerson = 25;
    private bool fancyDecorations;
    public decimal CostOfDecorations = 0;

    public Party(int numberOfPeople, bool fancyDecorations) {
        this.fancyDecorations = fancyDecorations;
        this.NumberOfPeople = numberOfPeople;
    }

    private int numberOfPeople;
    public virtual int NumberOfPeople {
        get { return numberOfPeople; }
        set {
            numberOfPeople = value;
            CalculateCostOfDecorations(fancyDecorations);
        }
    }

    public void CalculateCostOfDecorations(bool fancy) {
        fancyDecorations = fancy;
        if (fancy)
            CostOfDecorations = (NumberOfPeople * 15.00M) + 50M;
        else
            CostOfDecorations = (NumberOfPeople * 7.50M) + 30M;
    }

    public virtual decimal CalculateCost() {
        decimal TotalCost = CostOfDecorations + (CostOfFoodPerPerson * NumberOfPeople);
        if (NumberOfPeople > 12)
        {
            TotalCost += 100M;
        }
        return TotalCost;
    }
}
```

这些代码直接从DinnerParty和BirthdayParty类移出放到Party中。

原先DinnerParty和BirthdayParty构造函数中都要做的所有事情现在会在Party构造函数中完成。

NumberOfPeople必须是虚属性（virtual），因为BirthdayParty要覆盖这个属性（从而当人数改变时会计算新的蛋糕尺寸）。

装饰费用计算在生日聚会和宴会中是相同的，所以把它移到Party中很合理。这样一来，就不会在多个类中出现重复代码。

花费的计算必须是一个虚方法，因为生日聚会要覆盖这个方法（并通过调用基类的方法来扩展）。

```
class BirthdayParty : Party {
    public int CakeSize;

    public BirthdayParty(int numberOfPeople, bool fancyDecorations, string cakeWriting)
        : base(numberOfPeople, fancyDecorations) {
        CalculateCakeSize();
        this.CakeWriting = cakeWriting;
        CalculateCostOfDecorations(fancyDecorations);
    }

    private void CalculateCakeSize() {
        if (NumberOfPeople <= 4)
            CakeSize = 8;
        else
            CakeSize = 16;
    }

    private string cakeWriting = "";
    public string CakeWriting {
        get { return this.cakeWriting; }
        set {
            int maxLength;
            if (CakeSize == 8)
                maxLength = 16;
            else
                maxLength = 40;
            if (value.Length > maxLength) {
                MessageBox.Show("Too many letters for a " + CakeSize + " inch cake");
                if (maxLength > this.cakeWriting.Length)
                    maxLength = this.cakeWriting.Length;
                this.cakeWriting = cakeWriting.Substring(0, maxLength);
            } else
                this.cakeWriting = value;
        }
    }

    public override decimal CalculateCost() {
        decimal CakeCost;
        if (CakeSize == 8)
            CakeCost = 40M + CakeWriting.Length * .25M;
        else
            CakeCost = 75M + CakeWriting.Length * .25M;
        return base.CalculateCost() + CakeCost;
    }

    public override int NumberOfPeople {
        get { return base.NumberOfPeople; }
        set {
            base.NumberOfPeople = value;
            CalculateCakeSize();
            this.CakeWriting = cakeWriting;
        }
    }
}
```

构造函数依赖基类完成大多数工作。然后调用CalculateCakeSize()，就像原来的BirthdayParty构造函数中一样。

CalculateCakeSize()方法是生日聚会特定的方法，所以保留在BirthdayParty类中。

CakeWriting属性同样保留在BirthdayParty类中。

CalculateCost()也需要覆盖，因为它需要先计算蛋糕的费用，然后把蛋糕费用增加到Party类CalculateCost()方法所计算的花费，得到最终费用。

NumberOfPeople属性必须覆盖Party类中的相应属性，因为设置存取方法需要重新计算蛋糕的尺寸。设置存取方法要调用base.NumberOfPeople来执行基类Party中的设置存取方法。

待续，转256页。

干得不错！



**Exercise Solution**

上接255页

这是Kathleen解决方案中的最后一个类（窗体代码不做任何修改）。

这个公共字段只在宴会中使用，而生日聚会中不用，所以仍留在这个类中。

```
class DinnerParty : Party
{
    public decimal CostOfBeveragesPerPerson;

    public DinnerParty(int numberOfPeople, bool healthyOption,
                       bool fancyDecorations)
        : base(numberOfPeople, fancyDecorations) {
        SetHealthyOption(healthyOption);
        CalculateCostOfDecorations(fancyDecorations);
    }

    public void SetHealthyOption(bool healthyOption) {
        if (healthyOption)
            CostOfBeveragesPerPerson = 5.00M;
        else
            CostOfBeveragesPerPerson = 20.00M;
    }

    public decimal CalculateCost(bool healthyOption) {
        decimal totalCost = base.CalculateCost()
                          + (CostOfBeveragesPerPerson * NumberOfPeople);

        if (healthyOption)
            return totalCost * .95M;
        else
            return totalCost;
    }
}
```

要完成原来DinnerParty类所做的工作，新构造函数要调用Party的构造函数，然后调用SetHealthyOption()。

SetHealthyOption()方法保持不变。

DinnerParty需要一个不同的CalculateCost()（有一个参数），所以并不是覆盖这个方法，这里要进行重载。它使用base关键字调用Party中的CalculateCost()方法，然后增加饮料的花销，另外还要考虑到健康型选择的相应折扣。

我们将在第8章讨论重载是怎么回事。




**唉呀，程序里还有一个潜在的bug！**

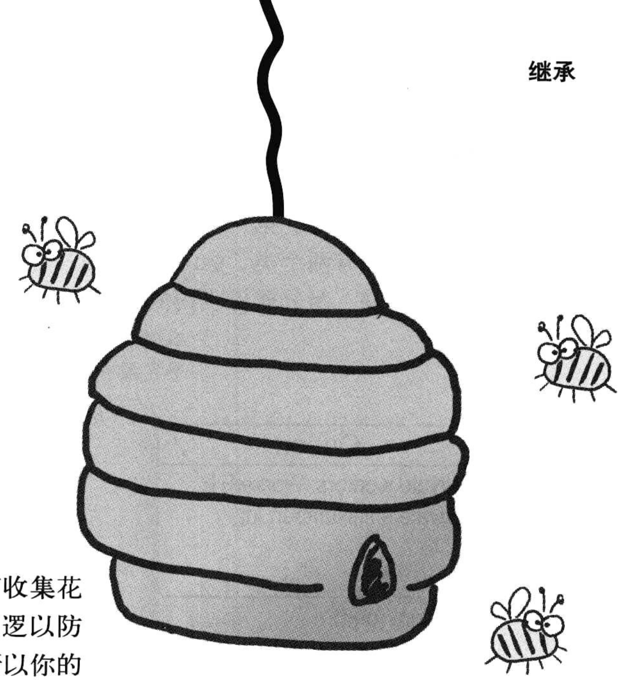
现在DinnerParty类有两个CalculateCost()方法，一个继承自Party，还是一个就是我们刚增加的这个新方法。我们还没有充分封装这个类，可能有人很容易误用这个代码，调用不合适的CalculateCost()方法。所以如果有下面的代码：

```
DinnerParty dinner = new DinnerParty(5, true, true);
decimal cost1 = dinner.CalculateCost(true);
decimal cost2 = dinner.CalculateCost();
```

cost1会设置为261.25，而cost2将设置为250。这不是一个形式上的问题，而是一个实实在在的问题。有时你并不想直接调用基类中的一些代码。更糟糕的是，我们根本不希望Party类得到实例化……不过没有办法禁止别人这么做。我们甚至不知道倘若真的有人创建了Party的一个实例会发生什么。不过可以确定的是，它肯定会做出一些预料之外的事情。

幸运的是，C#对于这些问题提供了一个很好的解决方案，有关内容将在下一章学习！

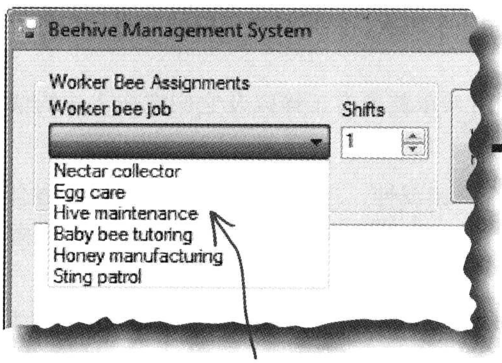
# 构建一个蜂巢管理系统



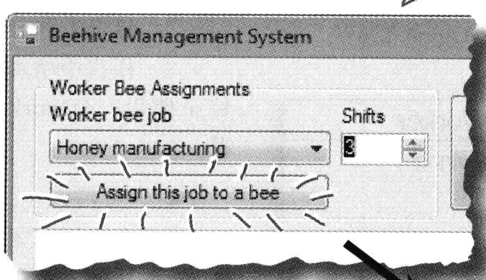
蜂王需要你的帮助！她的蜂巢失控了，需要一个程序来帮忙管理。蜂巢里满是工蜂，还有一大堆工作需要完成。不过她已经无法控制了，不知道哪只蜜蜂在做什么，也不知道有没有尽可能充分利用每只蜜蜂，让它们各尽其能。

现在要由你来建立一个蜂巢管理系统，帮助她跟踪所有工蜂的情况。系统的工作应该如下：

1. **蜂王向工蜂指派工作。**
   有6种可能的工作让工蜂去做。有些工蜂知道如何收集花露和酿造蜂蜜，另外一些可以维护蜂巢，还能巡逻以防外来入侵。有些蜜蜂可以做蜂巢里的所有工作。所以你的程序需要让为蜂王有办法为所有有能力的蜜蜂分派一个工作。

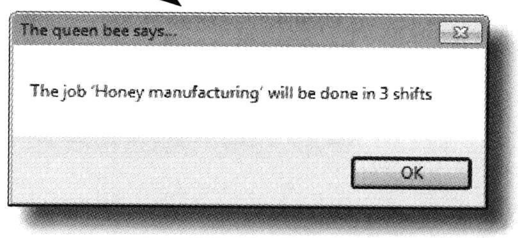



这个下拉表显示了工蜂能做的所有6项工作。蜂王知道哪些工作需要完成，她并不关心各个工作究竟由哪只蜜蜂来完成。所以她只是选择必须完成哪个任务，程序会明确有没有工蜂可以做这个工作，并把任务交给它。

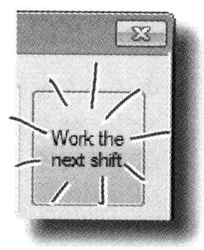



蜜蜂的工作需要轮班，而且大多数工作都需要多个班次才能完成。因此蜂王要输入完成任务的班次数，并单击"Assign this job"（分配这个任务）按钮。

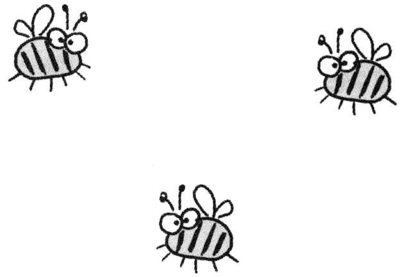
如果有蜜蜂可以做这个工作，程序将把任务分派给这个蜜蜂，并让蜂王知道这个任务正在完成。




2. **所有任务都已经分配后，下面开始工作。**
   一旦蜂王分配了所有工作，通过单击"Work the next shift"（换班）按钮她会告诉蜜蜂们换下一班。然后程序生成一个班次报告，告诉她这一班有哪些蜜蜂工作，完成了哪些工作，另外每个任务还需要多少班次才能完成。

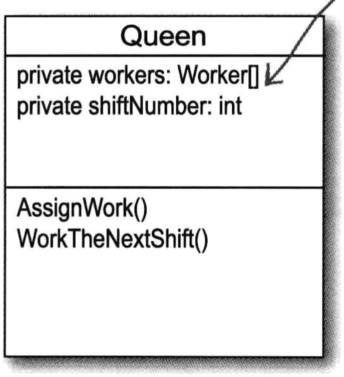


# 首先构建基本系统



这个工程划分为两部分。第一部分可以算是复习，将创建一个基本系统来管理蜂巢。它有两个类，Queen和Worker。要为系统建立窗体，并与这两个类关联。另外确保类得到很好的封装，以便到第二部分时易于修改。

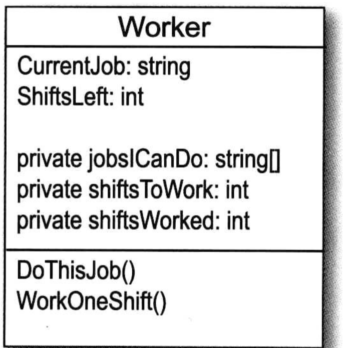
有时类图会列出私有字段和类型。




这个程序有一个Queen对象来管理正在完成的工作。

- Queen使用一个Worker对象数组来跟踪各个工蜂，查看这些蜜蜂是否已经分配了任务。这个数组存储在一个私有的Worker[ ]字段中，名为worker。
- 窗体调用AssignWork()方法，传入一个串表示需要完成的工作，另外传入一个int表示班次数。如果发现一个可以分配任务的蜜蜂，则返回true，如果找不到能完成这个任务的工蜂，则返回false。
- 窗体的“Work the next shift”按钮调用WorkTheNextShift()，通知工蜂工作，并返回一个班次报告显示出来。这个方法通知每个Worker对象工作一个班次，然后检查工蜂的状态，在班次报告中增加一行。

CurrentJob和ShiftsLeft都是只读属性。




蜂王使用一个Worker对象数组来跟踪所有工蜂以及它们完成工作的情况。

- CurrentJob是一个只读属性，告诉Queen对象工蜂正在做什么工作(“带刺巡逻”、“蜂巢维护”等等）。如果工蜂没有做任何工作，则返回一个空串。
- Queen对象尝试使用其DoThisJob()方法为一个工蜂分配一个工作。如果这个工蜂目前没有做这个工作，而且它确实知道怎么完成这个任务，就会接受任务，这个方法返回true。否则，返回false。
- 调用WorkOneShift()方法时，这个工蜂会工作一个班次。它会跟踪当前任务还需要多少班次才能完成。如果任务已经完成，则重置它的当前任务为一个空串，从而能接受下一个新的任务。

## String.IsNullOrEmpty()

由于每个蜜蜂都会把当前任务存储为一个串，工蜂要想知道它目前是否在做某个工作，就需要检查它的CurrentJob属性，如果它正在等待下一个任务，这个属性就等于一个空串。对此，C#提供了一个很容易的方法：如果CurrentJob串为空或为null，String.IsNullOrEmpty(CurrentJob)就会返回true，否则返回false。

**Exercise**

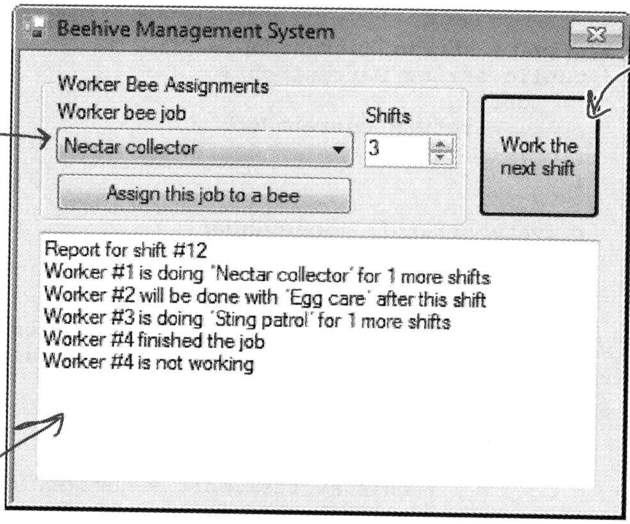
蜂王需要你的帮助！使用前面关于类和对象学到的知识建立一个蜂巢管理系统，帮助她跟踪工蜂们的工作。

**1** **建立窗体。**

窗体很简单，关键内容都在Queen和Worker类中。窗体有一个私有Queen字段，另外有两个按钮调用其AssignWork()和WorkTheNextShift()方法。需要增加一个ComboBox控件列出蜜蜂完成的任务（翻回到前一页，看看其中的列表项），再增加一个NumericUpDown控件、两个按钮和一个多行文本显示班次报告。另外窗体还需要一个构造函数（在以下截屏图下面）。

这是一个ComboBox控件，名为workerBeeJob。使用其Items属性来设置列表，将DropDownStyle属性设置为"DropDownList"，这样用户只能从列表中选择项（而不能另外输入）。Shifts框是一个NumericUpDown控件，名为"shifts"。

nextShift按钮调用蜂王的WorkTheNextShift()方法，这会返回一个串，其中包含班次报告。




仔细查看Queen对象生成的这个班次报告。首先是一个班次号，然后报告各个工蜂在做什么。这里使用转义序列"\r\n"在串中增加换行符。

将这个文本框命名为"report"，并设置其MultiLine属性为true。

```
public Form1() {
    InitializeComponent();
    Worker[] workers = new Worker[4];
    workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" });
    workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" });
    workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" });
    workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
            "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" });
    queen = new Queen(workers);
}
```

每个Worker对象的构造函数取一个串数组作为参数，指出它会做哪些工作。

窗体需要一个名为queen的Queen字段。要将Worker对象引用数组传入Queen对象的构造函数。

**2** **建立Worker和Queen类。**

以上基本上就是关于Worker和Queen类所需知道的全部内容。不过还有几个小细节需要说明。Queen.AssignWork()会循环处理Queen对象的worker数组，并试图使用其DoThisJob()方法为各个Worker分配任务。Worker对象检查自己的jobsICanDo串数组，查看自己能不能完成这个任务。如果可以，则设置其私有shiftsToWork字段为完成任务需要的班次数，将CurrentJob设置为这个任务，shiftNumber设置为0。工作一个班次时，将shiftNumber增1。只读的ShiftsLeft属性会返回shiftsToWork-shiftsWorked，蜂王使用这个属性来查看完成这个任务还需要多少班次。

**Exercise Solution**

构造函数只是设置JobsICanDo属性，这是一个串数组。它是私有的，因为我们希望蜂王只是要求工蜂做某个工作，而不是由蜂王来检查工蜂是否知道如何完成某个工作。

ShiftsLeft是一个只读属性，计算完成当前任务还需要多少班次。

CurrentJob是一个只读属性，告诉蜂王需要完成哪个任务。

蜂王使用工蜂的DoThisJob()方法为它分配工作，工蜂检查看自己的JobsICanDo属性，查看自己是否知道如何完成这个任务。

我们用了!操作符（逻辑非或NOT操作符）来检查串是否不为null或不为空。这就类似于检查某个条件是否为false。

蜂王使用工蜂的WorkOneShift()方法告诉它轮下一班。只有当这是完成这个任务的最后一班时，这个方法才会返回true。如果确实如此，蜂王可以在报告中增加一行，指出这一班结束后这个蜜蜂的任务就完成了。

请仔细看这里的逻辑。首先要检查currentJob字段：如果工蜂目前没有做任何工作，将返回false，方法会提前结束。如果并非如此，则让shiftsWorked增1，然后与ShiftsToWork比较，查看这个工作是否已经完成。如果确实已经完成，方法将返回true。否则返回false。

```
class Worker {
    public Worker(string[] jobsICanDo) {
        this.jobsICanDo = jobsICanDo;
    }

    public int ShiftsLeft {
        get {
            return shiftsToWork - shiftsWorked;
        }
    }

    private string currentJob = "";
    public string CurrentJob {
        get {
            return currentJob;
        }
    }

    private string[] jobsICanDo;
    private int shiftsToWork;
    private int shiftsWorked;

    public bool DoThisJob(string job, int numberOfShifts) {
        if (!String.IsNullOrEmpty(currentJob))
            return false;
        for (int i = 0; i < jobsICanDo.Length; i++)
            if (jobsICanDo[i] == job) {
                currentJob = job;
                this.shiftsToWork = numberOfShifts;
                shiftsWorked = 0;
                return true;
            }
        return false;
    }

    public bool WorkOneShift() {
        if (String.IsNullOrEmpty(currentJob))
            return false;
        shiftsWorked++;
        if (shiftsWorked > shiftsToWork) {
            shiftsWorked = 0;
            shiftsToWork = 0;
            currentJob = "";
            return true;
        }
        else
            return false;
    }
}
```

```
class Queen {
    public Queen(Worker[] workers) {
        this.workers = workers;
    }

    private Worker[] workers;
    private int shiftNumber = 0;

    public bool AssignWork(string job, int numberOfShifts) {
        for (int i = 0; i < workers.Length; i++)
            if (workers[i].DoThisJob(job, numberOfShifts))
                return true;
        return false;
    }

    public string WorkTheNextShift() {
        shiftNumber++;
        string report = "Report for shift #" + shiftNumber + "\r\n";
        for (int i = 0; i < workers.Length; i++)
        {
            if (workers[i].WorkOneShift())
                report += "Worker #" + (i + 1) + " finished the job\r\n";
            if (String.IsNullOrEmpty(workers[i].CurrentJob))
                report += "Worker #" + (i + 1) + " is not working\r\n";
            else
              if (workers[i].ShiftsLeft > 0)
                report += "Worker #" + (i + 1) + " is doing '" + workers[i].CurrentJob
                    + "' for " + workers[i].ShiftsLeft + " more shifts\r\n";
              else
                report += "Worker #" + (i + 1) + " will be done with '"
                    + workers[i].CurrentJob + "' after this shift\r\n";
        }
        return report;
    }
}
```

蜂王将其工蜂数组置为私有，因为一旦为工蜂分配了工作，其他类就不能再修改这些工蜂对象……甚至无法看到这些对象，这是因为只有蜂王能下命令。构造函数会设置这个字段的值。

为工蜂分配工作时，蜂王首先从第一只开始，尝试为它分配工作。如果这个工蜂没有能力完成这个工作，就转向下一只。找到一只能完成工作的蜜蜂时，方法返回（这会终止循环）。

蜂王的WorkTheNextShift()方法告诉每个工蜂轮下一班，并根据工蜂的状态向报告增加一行输出。

我们已经给出了构造函数。这是窗体的其余代码：

窗体使用其queen字段来保存Queen对象的一个引用，这个对象包含一个worker对象引用数组。

```
Queen queen;

private void assignJob_Click(object sender, EventArgs e) {
    if (queen.AssignWork(workerBeeJob.Text, (int)shifts.Value) == false)
        MessageBox.Show("No workers are available to do the job '"
            + workerBeeJob.Text + "'", "The queen bee says......");
    else
        MessageBox.Show("The job '" + workerBeeJob.Text + "' will be done in "
            + shifts.Value + " shifts", "The queen bee says......");
}

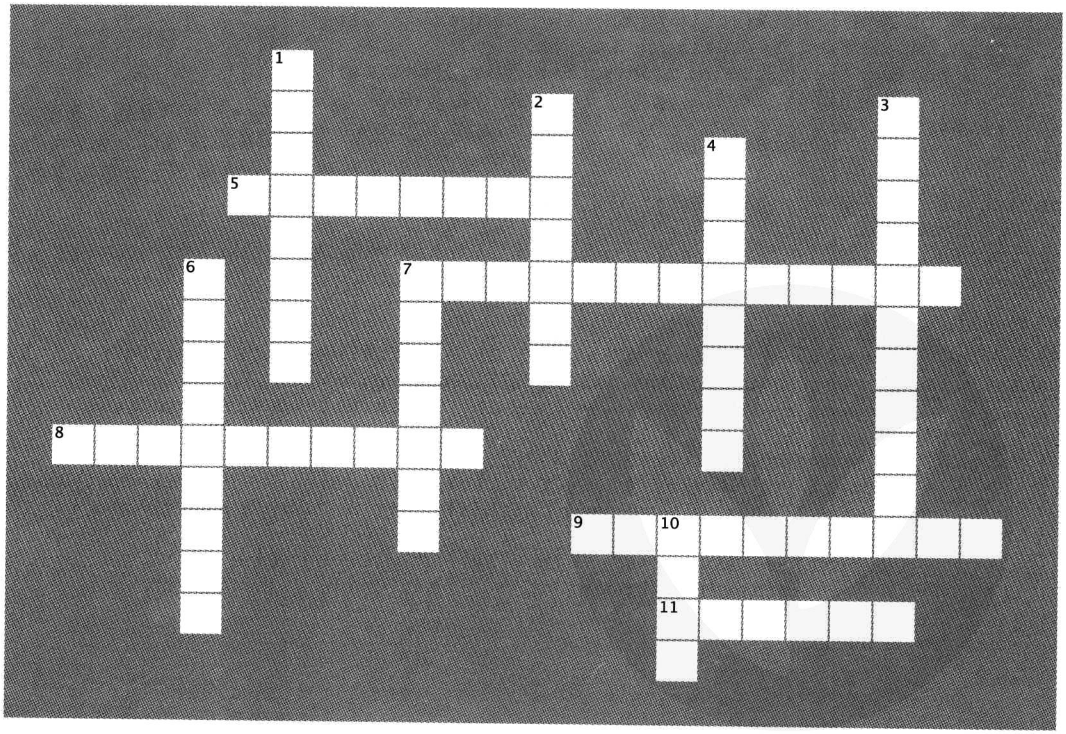
private void nextShift_Click(object sender, EventArgs e) {
    report.Text = queen.WorkTheNextShift();
}
```

assignJob按钮调用蜂王的AssignWork()方法为一个工蜂分配工作，并根据是否有工蜂能够完成这个任务显示一个消息框。

nextShift按钮告诉蜂王换下一班。她会生成一个报告，并在report文本框中显示。

# 继承填字游戏

转向练习的下一部分之前，先让大脑休息一下，做一个简单的填字游戏。




## 横向

5.这个方法获得一个属性的值。

7.如果传入“”，则这个方法将返回true。

8.子类中的构造函数不需要与其基类中的构造函数有相同的____________。

9.这是窗体中的一个控件，可以用于创建分页应用。

11.这种类型的类不能实例化。

## 纵向

1.____________可以覆盖其基类中的方法。

2.如果希望一个子类覆盖一个方法，则要对基类中的这个方法用____________关键字标志。

3.一旦实例化类就要运行类中的这个方法。

4.子类替换基类中的一个方法称为____________。

6.____________包含基类和子类。

7.向类声明增加一个冒号时所做的工作。

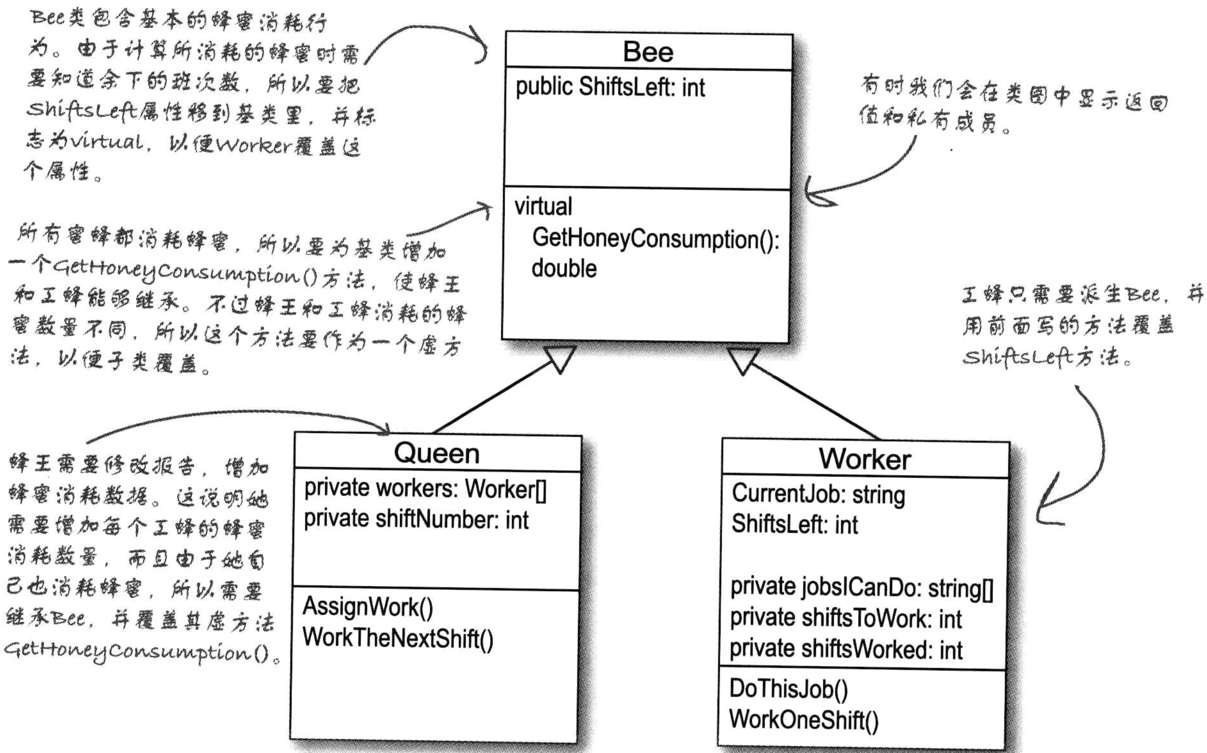
10.子类使用这个关键字调用它继承的父类的成员。

答案见268页。

# 使用继承扩展蜜蜂管理系统



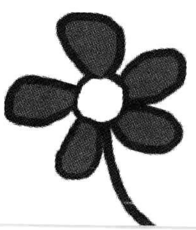
既然已经有了这样一个基本的系统，下面就再使用继承来跟踪每个蜜蜂消耗多少蜂蜜。不同蜜蜂消耗的蜂蜜数量不同，蜂王消耗的蜂蜜最多。所以下面根据前面学到的继承知识创建一个Bee基类，Queen和Worker都要继承这个基类。

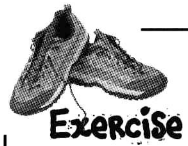



## 增加现有项

如果要完成的练习包括两个部分，则我们有一个好办法，可以为第二部分另外启动一个新工程。这样一来，需要时就可以返回到第一个解决方案（第一部分）。要做到这一点有一种简单的做法，在IDE中，可以在新工程的Solution Explorer中鼠标右键单击工程名，从菜单选择"Add Existing Item"导航到老工程的文件夹。选择想要增加的文件。IDE会在新工程的文件夹中建立这些文件的新副本，并把它们增加到工程中。不过有几个问题需要注意。IDE不会改变命名空间，所以需要编辑各个类文件，手动地修改其命名空间行。如果增加了窗体，要确保增加其设计文件(.Designer.cs)和资源文件（.resx），另外同样要修改这两个文件的命名空间。








我们的工作还没有完成!蜂王需要跟踪她的工蜂消耗了多少蜂蜜。这是一个很好的机会，可以借此施展你新学的继承技术!

**❶ 蜂王需要知道蜂巢总共使用了多少蜂蜜。**

蜂王刚刚收到“会计蜂”的来电，告诉她蜂巢产蜜量有些不足。她要知道她自己和工蜂们消耗了多少蜂蜜，才能决定是不是需要把一些负责养护蜂卵的工蜂换去完成酿蜜的工作。

- ★ 所有蜜蜂都要吃蜜，所以蜂巢要用掉很多蜜。正因如此它们需要产更多的蜜。
- ★ 工蜂工作时吃掉的蜜更多。任务刚开始时它们需要的蜜最多，以便提供足够的能量来完成任务。随着任务的进行，消耗的蜂蜜越来越少。到最后一班，一只蜜蜂只使用10个单位的蜂蜜，倒数第二班需要11个单位，这之前的一班（倒数第三班）要使用12个单位，以此类推。所以，如果一个蜜蜂在工作（这说明它的ShiftsLeft大于0），则可以将ShiftsLeft加9来得出它要消耗多少蜂蜜。
- ★ 如果一个蜜蜂没有工作（也就是说，其ShiftsLeft为0），则等待工作期间只使用7.5个单位的蜂蜜。
- ★ 这些数字都只是针对普通的蜜蜂。如果一个蜜蜂重量超过150毫克，则会多使用35%的蜜。不过这不包括蜂王（见下面的说明）。
- ★ 蜂王需要大量的蜂蜜。蜂王为工蜂指派工作时，工蜂越多，蜂王使用的蜜也越多，因为监督它们的工作很多。蜂王消耗的蜂蜜量等同于工作最繁重的工蜂，就像剩余班次最多的工蜂一样。
- ★ 蜂王还需要额外的一些蜂蜜：如果工作的工蜂只有2个或少于2个，则每一班次蜂王要多用20个单位的蜂蜜，如果有3或3只以上工蜂在工作，那么蜂王还需要另外30个单位的蜂蜜。蜂王的蜂蜜消耗量不受35%规则的影响，因为所有蜂王体重都达到275毫克。
- ★ 蜂王需要将所有蜂蜜消耗量增加到每个班次报告的最后。

**❷ 创建一个Bee类处理蜂蜜消耗量的计算。**

由于工蜂和蜂王会以类似的方式计算蜂蜜消耗量，所以可以建立一个`Bee`基类，让`Worker`和`Queen`继承这个类，从而避免代码重复。你很清楚，每个蜜蜂都需要知道自己的重量（从而知道是否需要将其蜂蜜消耗量乘以35%）。

- ★ 创建一个`GetHoneyConsumption()`方法，计算工蜂使用的蜂蜜量。工蜂和蜂王都需要做这个计算，不过蜂王还需要完成额外的一些计算，所以合理的做法是让工蜂继承这个方法，而让蜂王覆盖这个方法。
- ★ `GetHoneyConsumption()`方法需要知道余下的班次数，所以增加一个只读的虚属性，名为`ShiftsLeft`，它返回0。工蜂的`ShiftsLeft`要覆盖这个属性。
- ★ 计算蜂蜜消耗量需要知道蜜蜂的重量，所以`Bee`构造函数要取重量作为参数，并把它存储在一个字段中。由于其他类都不需要使用这个字段，所以应当把这个字段置为私有字段。



这是一个很好的经验。默认地应当声明字段和方法为私有，只有当另一个类需要它们时才置其为公共。这样一来，可以避免程序中由于一个类不适当地访问另一个类的属性或方法而导致的bug。

提示：可以充分利用有些费解的“no overload”错误消息！让Worker类继承Bee，然后构建工程。IDE显示“no overload”错误时，双击这个错误，IDE会立即自动跳至Worker构造函数。多方便！



## ❸ 让Worker类继承Bee。

需要建立构造函数来调用基类的构造函数，就像在Kathleen的程序中一样。要修改Worker构造函数，取蜜蜂的重量作为参数，并把这个参数传递到基类构造函数。然后，只需为Worker的ShiftLeft方法增加override关键字。一旦做到这一点，每个工蜂就能为蜂王计算蜂蜜消耗量了…… 不必对Worker类做任何修改！

## ❹ 让Queen类继承自Bee。

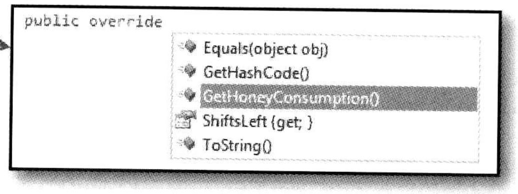
与Worker类相比，对Queen类做的修改稍多一点，因为蜂王需要具体完成蜂蜜消耗量计算，并增加到班次报告中。

★ 覆盖Bee.GetHoneyConsumption()方法，并增加蜂王额外完成的计算。她需要得出是否有2个或更少的工蜂在工作，从而知道额外需要的蜂蜜量（20或30个单位）。然后将这个额外消耗的蜂蜜量与正常消耗量相加，蜂王的正常消耗量就相当于剩余班次最多的工蜂所消耗的蜂蜜。

★ 更新蜂王的WorkTheNextShift()，将蜂蜜消耗量增加到报告中。增加一个循环，累计各个工蜂的蜂蜜消耗数量，另外找出蜂蜜消耗量最大的工蜂。要在蜂王告诉每个工蜂轮班之前做这个工作（这样她就能得到当前班次的消耗量）。蜂王把这些都加起来，并加上她自己的消耗量，再在班次报告的最后增加一行，指出“总共消耗蜂蜜：xxx单位”（其中xxx是指消耗了多少个单位的蜂蜜）。

★ 与Worker中一样，还要以同样的方式更新Queen的构造函数。

在Queen类中，输入“public override”，按空格时，IDE会自动列出可以覆盖的所有方法。选择你想覆盖的那个方法，其中会自动填入相应的基类方法调用。




## ❺ 更新窗体，正确地实例化蜜蜂。

由于修改了Queen和Worker构造函数，还需要修改调用它们的方式。每个构造函数有一个新的Weight参数，所以需要使用重量：

★ Worker Bee #1: 175mg; Worker Bee #2: 114mg; Worker Bee #3: 149mg; Worker Bee#4: 155mg; Queen Bee: 275mg

这就是对窗体唯一要做的修改！

**Exercise Solution**

以下是Bee类。它完成Worker和Queen都要使用的基本蜂蜜消耗量计算。

```
class Bee {
    public Bee(double weight) {
        this.weight = weight;
    }

    public virtual int ShiftsLeft {
        get { return 0; }
    }

    private double weight;

    public virtual double GetHoneyConsumption() {
        double consumption;
        if (ShiftsLeft == 0)
            consumption = 7.5;
        else
            consumption = 9 + ShiftsLeft;
        if (weight > 150)
            consumption *= 1.35;
        return consumption;
    }
}
```

Bee类有一个构造函数来设置其Weight字段，另外有一个HoneyConsumption()方法计算一个工蜂消耗多少蜂蜜。

如果一个蜜蜂还剩1班，则消耗10个单位的蜂蜜；如果还剩2班，则消耗11，以此类推。如果没有任务，则消耗7.5。如果ShiftsLeft为0，说明这个蜜蜂没有任务。

如果蜜蜂重量超过150mg，则蜂蜜消耗量会增加35%。

**利用继承，可以很容易地更新代码，为Queen和Worker类增加新的蜂蜜消耗行为。如果有很多重复代码，完成这种修改就会困难得多。**

只有窗体构造函数有变化，窗体的余下部分完全不变。

```
public Form1() {
  InitializeComponent();

  Worker[] workers = new Worker[4];
  workers[0] = new Worker(new string[] { "Nectar collector", "Honey manufacturing" }, 175);
  workers[1] = new Worker(new string[] { "Egg care", "Baby bee tutoring" }, 114);
  workers[2] = new Worker(new string[] { "Hive maintenance", "Sting patrol" }, 149);
  workers[3] = new Worker(new string[] { "Nectar collector", "Honey manufacturing",
            "Egg care", "Baby bee tutoring", "Hive maintenance", "Sting patrol" }, 155);
  queen = new Queen(workers);
}
```

只需对窗体做一处修改，向Worker构造函数增加重量。

```
class Worker : Bee    {
    public Worker(string[] jobsICanDo, int weight)
        : base(weight) {
        this.jobsICanDo = jobsICanDo;
    }
    public override int ShiftsLeft {
    // ...the rest of the class is the same ......
```

Worker类需要做的就是继承Bee，并调整它的构造函数，要求有一个Weight参数，并将这个参数传入基类构造函数，另外还要覆盖Bee.ShiftsLeft属性，在该属性声明前增加override关键字。

Queen类需要做一些修改，先要继承Bee。

```
class Queen : Bee {
    public Queen(Worker[] workers)
        : base(275) {
        this.workers = workers;
    }

    public string WorkTheNextShift()
    {
        double totalConsumption = 0;
        for (int i = 0; i < workers.Length; i++)
            totalConsumption += workers[i].GetHoneyConsumption();
        totalConsumption += GetHoneyConsumption();

        // ...here's where the original code for this method goes, minus the return statement

        report += "Total honey consumption: " + totalConsumption + " units";
        return report;
    }
    public override double GetHoneyConsumption() {
        double consumption = 0;
        double largestWorkerConsumption = 0;
        int workersDoingJobs = 0;
        for (int i = 0; i < workers.Length; i++) {
            if (workers[i].GetHoneyConsumption() > largestWorkerConsumption)
                largestWorkerConsumption = workers[i].GetHoneyConsumption();
            if (workers[i].ShiftsLeft > 0)
                workersDoingJobs++;
        }
        consumption += largestWorkerConsumption;
        if (workersDoingJobs >= 3)
            consumption += 30;
        else
            consumption += 20;
        return consumption;
    }
}
```

蜂王重量为275mg，所以她的构造函数调用基类Bee的构造函数，并输入重量275。

WorkTheNextShift()在最上面增加了一个循环，调用各工蜂的GetHoneyConsumption()方法，然后调用她自己的GetHoneyConsumption()方法，得到总的消耗量。

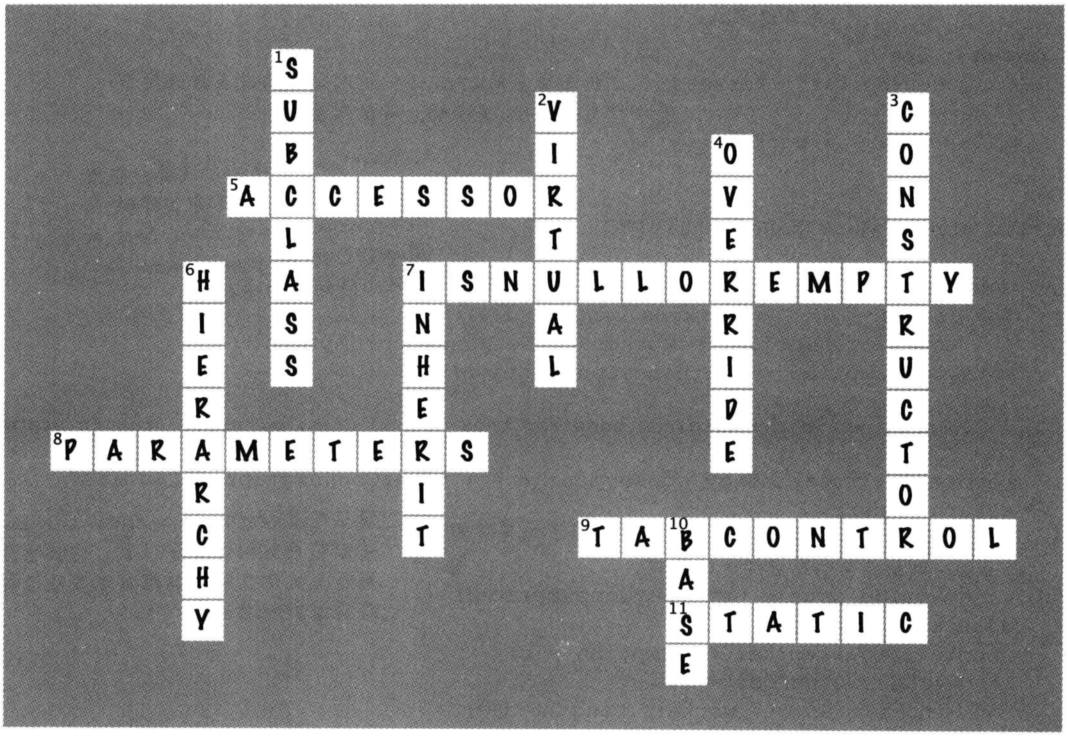
WorkTheNextShift()的余下部分都一样，只不过需要向报告增加有关蜂蜜消耗量的一行输出。

蜂王覆盖Bee的GetHoneyConsumption()方法来完成她的蜂蜜量计算。它找出蜂蜜消耗量最大的工蜂，并根据有多少工蜂在工作再增加20或30。

这个循环查看所有工蜂的蜂蜜消耗量，找出蜂蜜消耗量最大的工蜂。

如果有3个或更多工蜂在工作，蜂王另外还需要30个单位的蜂蜜。否则，需要再加20个单位。

# 继承填字游戏答案


1 SUBCLASS
2 VIRTUAL
3 CONSTRUCTOR
4 OVERRIDE
5 ACCESSOR
6 HIERARCHY
7 ISNULLOREMPTY
7 INHERIT
8 PARAMETERS
9 TABCONTROL
10 BASE
11 STATIC

# 7 接口与抽象类

# 让类信守承诺




**行动比言语更有力。**

有时需要根据对象完成的工作将对象分组，而不是根据它们继承的类来分组。这就引入了接口，通过接口，可以使用任何一个能完成任务的类。但是能力越大，责任也越大，实现了一个接口的类必须承诺履行它的所有职责……否则编译器就会抱怨，明白吗?

# 再来看蜂巢问题

统一Bee-namics公司想把上一章创建的蜂巢管理系统变成一个完备的蜂巢模拟系统。以下概要列出了这个新版本程序的规范：



## 统一Bee-namics蜂巢模拟系统

为了更好地表示蜂巢内的生活，需要为工蜂增加一些特殊的功能。

- 所有蜜蜂都要消耗蜂蜜，另外每个蜜蜂自身都有一定重量。
- 蜂王要分配工作，监控班次报告，通知工蜂轮班。
- 所有工蜂轮流换班。
- 带刺巡逻蜂需要擦尖它们的蜂刺，搜寻敌人，并拿刺蜇来犯的入侵者。
- 收集花露的蜜蜂负责寻找鲜花，收集花露，然后返回蜂巢。

看起来Bee和Worker类不用做太多修改。我们可以扩展已有的类来处理这些新特性。

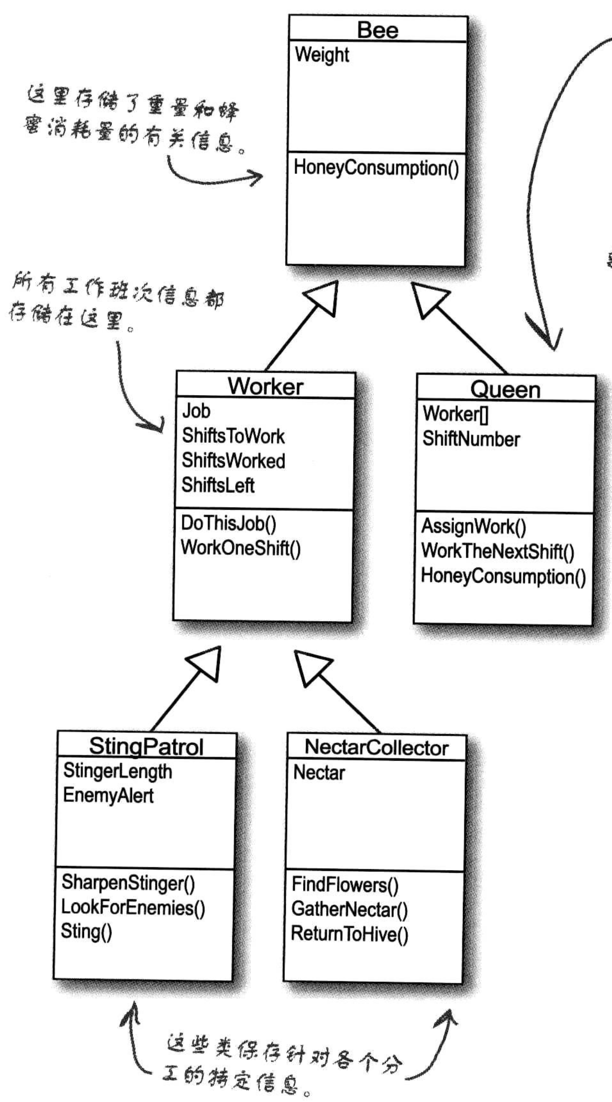
看上去需要根据工蜂的不同分工分别存储不同的数据。

## 很多方面仍保持不变。

这个新的蜂巢模拟系统中，蜜蜂还是会像以前一样消耗蜂蜜。蜂王仍然需要为工蜂分配工作，并查看班次报告来了解谁在做什么。工蜂轮班的方式也与以前很类似，只不过现在它们的分工更细一些。

# 可以使用继承为不同类型的蜜蜂创建相应的类

以下类层次体系中，Worker和Queen类继承自Bee，另外Worker又有子类NectarCollector和StingPatrol。




还记得蜂王需要额外的蜂蜜吗？要在这里覆盖HoneyConsumption()方法。

StingPatrol和NectarCollector继承自Worker类。

新子类如下所示。

```
class StingPatrol : Worker
{
     int StingerLength;
     bool enemyAlert;
     public bool SharpenStinger (int Length)
     { ... }
     public bool LookForEnemies(){ ... }
     public void Sting(string Enemy){ ... }
}
```

```
class NectarCollector : Worker
{
     int Nectar;
     public void FindFlowers (){ ... }
     public void GatherNectar(){ ... }
     public void ReturnToHive(){ ... }
}
```

**BRAIN POWER**

如果需要一个蜜蜂既能带刺巡逻又能收集花露，该怎么做呢？

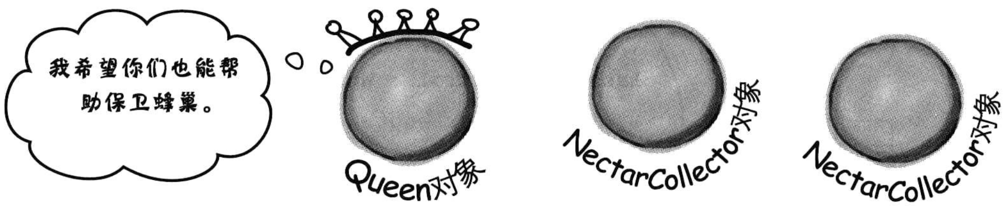
## 接口告诉类必须实现某些方法和属性

类只能继承另外的一个类。所以如果只是分别为StingPatrol和NectarCollector蜜蜂创建两个单独的类，对于一个两样工作都能做的蜜蜂（既能带刺巡逻又能收集花露）将于事无补。

蜂王的DefendTheHive()方法只能通知StingPatrol对象确保蜂巢的安全。她很愿意训练其他蜜蜂使用它们的蜂刺，不过没有办法命令它们主动出击。

**使用接口要求一个类包含接口中所列的全部方法和属性。如果类没有做到这一点，则编译器会报错。**

```
class Queen {
  private void DefendTheHive(StingPatrol patroller) { ... }
}
```




NectarCollector对象知道如何从花收集花露，还有一些StingPatrol实例可以擦尖它们的蜂刺，巡逻搜寻敌人。不过即使蜂王可以在类定义中增加类似SharpenStinger()和LookForEnemies()的方法教NectarCollector保卫蜂巢，她也无法将NectarCollector传入她的DefendTheHive()方法。她可以使用两个不同的方法：

```
private void DefendTheHive(StingPatrol patroller);
private void AlternateDefendTheHive(NectarCollector patroller);
```

尽管蜂王为NectarCollector对象增加了带刺巡逻方法，但还是不能把NectarCollector对象传入她的DefendTheHive()方法，因为这个方法需要一个StingPatrol引用。她不能简单地设置StingPatrol引用等于一个NectarCollector对象。

她也可以增加另一个名为AlternateDefendTheHive()的方法，取一个NectarCollector引用作为参数，不过这样很麻烦，而且使用很困难。

而且，DefendTheHive()和AlternateDefendTheHive()方法基本上是相等的，只不过参数的类型有所不同。如果蜂王想教BabyBeeCare或Maintenance对象保卫蜂巢，她就需要继续增加新方法。那该多混乱呀！

不过这并不是一个很好的解决方案。这两个方法应该是相等的，因为它们都会对所传入的对象调用同样的方法。唯一的区别是其中一个方法有一个StingPatrol参数，另一个方法需要一个NectarCollector对象（该NectarCollector对象要有在蜂巢巡逻所需的方法）。你应该已经知道维护两个相等的方法是多么痛苦。

幸运的是，C#提供了接口（interfaces）来处理这种情况。利用接口，可以定义一个类必须有的一组方法。

接口要求一个类必须有某些方法，而且这是强制性的，如果在实现某个接口的各个类中没有找到该接口所要求的所有方法，编译器就会报错。这些方法可以在类中直接编写，也可以从一个基类继承。只要编译代码时类中有接口所要求的方法和属性，接口并不关心这些方法或属性是如何得来的。

# 使用interface关键字定义接口

为程序增加接口与增加类是非常相似的，只不过不用编写任何方法。只需定义方法的返回类型和参数，另外不需要在大括号内编写语句，只要在这一行的最后加一个分号就可以了。

接口不存储数据，所以不能增加任何字段。不过可以增加属性定义。这是因为，获取或设置存取方法也是一种特殊类型的方法，而接口就是要求类中包含某些有特定名、类型和参数的方法。所以如果遇到这种问题，需要在接口中增加字段，就可以使用属性来达到目的，它会完成你想要做到的事情。

> **接口名以I开头**
>
> 只要创建接口，就应当让接口名以一个大写的I开头。对于这一点并没有明确的规定，但是这样可以使你的代码更易于理解。可以自己试试看，这样做肯定能让你的日子好过得多。只需在IDE中任何方法内的任何空行上输入“I”，就会看到智能提示（IntelliSense）窗口，其中会列出.NET接口。

声明这样一个接口：

```
interface IStingPatrol
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

接口不存储数据，所以没有字段…… 不过可以有属性。

实现这个接口的任何类都必须包含所有这些方法和属性，否则程序将不能编译。

实现这个接口的所有类都要有带一个int参数的SharpenStinger()方法。

```
interface INectarCollector
{
    void FindFlowers();
    void GatherNectar();
    void ReturnToHive();
}
```

不用为接口中的方法编写代码，只要有方法名就行了。要在实现这个接口的类中具体编写代码。

那么这对蜂王有什么帮助呢？现在她可以建立一个方法，取任何知道如何保卫蜂巢的对象作为参数：：

```
private void DefendTheHive(IStingPatrol patroller)
```

因为这里需要一个IStingPatrol引用，可以传入实现IStingPatrol的任何对象。

这样蜂王就只需要一个方法，这个方法可以取一个StingPatrol、NectarStinger和任何知道如何保卫蜂巢的其他对象作为参数，向这个方法传入哪个类并不重要。只要这个类实现了IStingPatrol，就可以向DefendTheHive()方法保证这个对象确实有保卫蜂巢所需的方法和属性。

**公共接口中的所有方法都会自动成为公共方法，因为接口就是用来定义实现该接口的类应该具有的公共方法和属性。**



# 现在可以创建两项工作都胜任的NectarStinger实例

类似于继承，要使用冒号操作符来实现接口。做法如下：冒号后首先是所继承的类，后面是一组接口（除非这个类没有继承任何类，在这种情况下，冒号后面只有一组接口，各接口没有特定的顺序）。

类似于继承，要使用冒号操作符来实现接口。

这个类继承了Worker，并实现了INectarCollector和IStingPatrol。

```
class NectarStinger : Worker, INectarCollector,
IStingPatrol {
        public int AlertLevel {
            get { return alertLevel; }
        }

        public int StingerLength {
            get { return stingerLength; }
            set {
                stingerLength = value;
            }
        }

        public bool LookForEnemies() { ... }
        public int SharpenStinger(int length)
        { ... }
        public void FindFlowers() { ... }
        public void GatherNectar() { ... }
        public void ReturnToHive() { ... }
    }
```

可以使用多个接口，不过要用逗号分隔。

NectarStinger在LookForEnemies()方法中设置AlertLevel属性的后备字段。

对应接口中的每个方法，在类中也要有相应的方法，否则将无法编译。

如果没有敌人，蜜蜂会收回蜂刺，所以后备字段的值会随时间改变。

创建一个NectarStinger对象时，它将身兼NectarCollector和StingPatrol工蜂的双重能力。

一个类实现了某个接口时，它与所有其他类是一样的。可以使用new实例化这个类，还可以使用它的方法：

```
NectarStinger bobTheBee = new NectarStinger();
bobTheBee.LookForEnemies();
bobTheBee.FindFlowers();
```

## there are no Dumb Questions

**问：** 我还是不清楚接口对蜂巢代码有什么改进。你仍然需要增加一个NectarStinger类，而且它还是有重复的代码……不是吗？

**答：** 接口并不是用来避免重复代码的。它们的作用是允许一个类在多种不同情况下使用。我们的目的是创建一个可以做两项不同工作的工蜂类。你仍然需要为它们创建类，这并不是重点。接口的关键在于：现在你已经有办法让一个类做多个工作。假设有一个PatrolTheHive()方法，它取一个StingPatrol对象，另外有一个CollectNectar()方法，取一个NectarCollector对象。不过你不希望StingPatrol继承NectarCollector，也不希望NectarCollector继承StingPatrol，每个类都有另一个类所没有的公共方法和属性。现在花点时间想个办法来创建一个类，使得这两个方法都可以传入这个类的实例。说真的，请把书放下，真正花点时间好好想想！该怎么做呢？

接口就能解决这个问题。现在你可以创建一个IStingPatrol引用。它可以指向实现了IStingPatrol的任何对象，而不论它具体是什么类。它可以指向一个StingPatrol，也可以是一个NectarStinger，甚至还可以是一个完全无关的对象。如果得到一个指向某个对象的IStingPatrol引用，你就会知道完全可以使用属于IStingPatrol接口的所有方法和属性，而不论对象的具体类型是什么。

不过接口还不是全部。还需要创建一个实现这个接口的新类，因为接口本身并没有提供任何代码。接口并不是用来避免创建额外的类或者避免重复代码。接口的目的是让一个类可以完成多个任务，而不依赖于继承，因为继承会带来很多额外的负担，你必须继承每一个方法、属性和字段，而不只是与处理特定任务有关的成员。

你能想出方法做到使用接口的同时还能避免重复代码吗？可以创建一个单独的类，名为Stinger或Proboscis，其中包含带刺巡逻或收集花露特定的代码。NectarStinger和NectarCollector都可以创建一个私有的Proboscis实例，只要需要收集花露，他们就可以调用其方法并设置其属性。

# 实现接口的类必须包括接口的所有方法

实现一个接口意味着，对于接口中声明的每一个属性和方法，在类中都要有一个相应的属性和方法，如果没有包括接口中的所有方法，就无法编译。如果一个类实现了多个接口，则需要包括它实现的每一个接口中的所有属性和方法。不过不要只是听我们讲，你可以自己动手试试看……

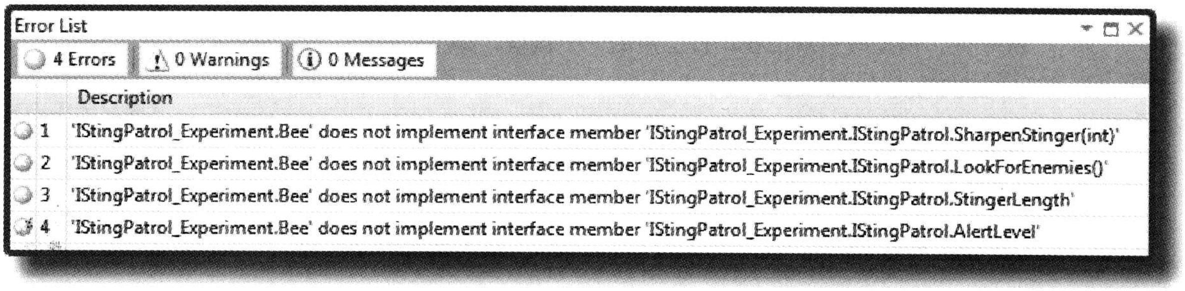
动手做！

1. **创建一个新应用，并增加一个新的类文件，名为IStingPatrol.cs。**
这里不增加类，而是输入前两页中的`IStingPatrol`接口。程序应该能编译了。

2. **为工程增加一个Bee类。**
先不要增加任何属性和方法。只是让这个类实现`IStingPatrol`：

```
class Bee : IStingPatrol
{
}
```

3. **尝试编译这个程序。**
从Build菜单选择“Rebuild”，唉呀，编译器不允许你这么做：




你会看到，如果类中没有实现IStingPatrol的每一个成员，对应每一个没有具体实现的IStingPatrol成员，都会显示一个这样的“does not implement”错误。编译器确确实实希望你实现接口中的每一个方法。

4. **向Bee类增加方法和属性。**
增加一个`LookForEnemies`方法和一个`SharpenStinger`方法，它们不用做任何具体的工作，增加这些方法只是因为要想完成编译就必须有这些方法。然后为一个名为`AlertLevel`的`int`属性增加一个获取存取方法，并为一个名为`StingerLength`的`int`属性增加获取和设置存取方法。现在程序就能编译了！

# 使用接口做个小练习

接口很容易使用，不过要想理解接口，最好的办法莫过于具体使用。所以创建一个新的Windows Forms Application工程，在窗体上拖入一个按钮，下面开始吧！

动手做！

1. 以下给出了TallGuy类，另外还显示了一个按钮的代码，其中使用对象初始化方法创建了一个TallGuy对象，并调用其TalkAboutYourself()方法。这里没有新内容，稍后将使用这个类：

```
class TallGuy {
    public string Name;
    public int Height;

    public void TalkAboutYourself() {
        MessageBox.Show("My name is " + Name + " and I'm "
            + Height + " inches tall.");
    }
}

private void button1_Click(object sender, EventArgs e) {
    TallGuy tallGuy = new TallGuy() { Height = 74, Name = "Jimmy" };
    tallGuy.TalkAboutYourself();
}
```

2. 下面为这个类创建一个IClown接口。

你已经知道，接口中的所有成员都必须是公共的。不过不要只听我们讲，你可以自己试一试。创建一个新工程，自己声明一个接口，如下：

```
interface IClown
```

现在尝试在这个接口中声明一个私有方法：

```
private void Honk();
```

在接口中不需要输入“public”，因为会自动将各个属性和方法置为公共。

在IDE中选择“Build”→“Build Solution”。你会看到以下错误：

> 1 The modifier 'private' is not valid for this item

再删除这个private访问修饰符，错误将消失，现在你的程序能正常编译了。

3. 读下一页之前，看看你自己能不能完成这个IClown接口的余下部分，并修改TallGuy类，让它实现这个接口。把这个接口增加到工程中，与增加类的做法相同，只需在Solution Explorer中鼠标右键单击工程，并增加一个名为IClown.cs的类文件。

这个新的IClown接口应该有一个名为Honk的void方法，它没有任何参数，另外接口中还有一个只读的string属性，名为FunnyThingIHave，它有一个获取存取方法但没有设置存取方法。

**4** 接口如下，你做对了吗？

```
interface IClown
{
    string FunnyThingIHave { get; }
    void Honk();
}
```

这是一个有获取存取方法而没有设置存取方法的属性。要记住，接口不能包含字段，但是如果在类中实现了这样一个只读属性，对于其他对象来说就像是一个字段。

好了，现在修改TallGuy类，让它实现IClown接口。记住，冒号操作符后面是这个类继承的基类（如果有），然后是要实现的一组接口，各接口之间用逗号分隔。由于没有基类，而且只需实现一个接口，所以声明如下：

```
class TallGuy : IClown
```

TallGuy将实现IClown接口。

然后确保类中的余下部分保持不变，包括两个字段和一个方法。在IDE中从Build菜单选择“Build Solution”，编译并构建这个程序。会看到两个错误，其中包括下面这个错误：

IDE在告诉你，既然你说TallGuy要实现IClown，就是在承诺会增加该接口中的所有属性和方法……但是你并没有信守承诺！

```
'TallGuy' does not implement interface
member 'IClown.Honk()'
```

**5** 一旦增加了接口中定义的所有方法和属性，这些错误就会消失。所以下面继续实现接口。再增加一个只读的string属性，名为FunnyThingIHave，它有一个获取存取方法，总返回串“big shoes”。然后增加一个Honk()方法，弹出一个显示“Honk honk!”的消息框。

代码如下：：

```
public string FunnyThingIHave {
    get { return "big shoes"; }
}
```

这个接口只要求实现该接口的类有一个名为FunnyThingIHave的属性，而且它有一个获取存取方法。在这里可以放任何获取存取方法，甚至获取存取方法可以每次返回相同的串。大多数获取存取方法不会这样做，但是如果你确实需要如此，这样一个获取存取方法也是允许的。

```
public void Honk() {
    MessageBox.Show("Honk honk!");
}
```

接口指出，需要一个名为Honk的公共void方法，不过它并没有说这个方法需要做什么。这个方法可以做任何事情，不论它做什么，只要存在这样一个有正确签名的方法，代码都能编译。

**6** 现在代码就能编译了！更新按钮的相应代码，调用TallGuy对象的Honk()方法。

# 不能实例化接口，不过可以引用接口

假设有一个方法需要一个对象作为参数，而且要求这个对象能够完成FindFlowers()方法。那么实现了INectarCollector接口的任何对象都可以满足要求。这可以是一个Worker对象，或者是一个Robot对象或Dog对象，只要它实现了INectarCollector接口。

这就引入了接口引用（interface reference）。可以使用一个接口引用指示实现了所需接口的一个对象，而且总能保证这个对象有适当的方法，可以用来达到你的目的（即使你对它并没有多少了解）。

你可以创建IWorker引用的一个数组，但是不能实例化接口。不过你能做到将这些引用指向实现了IWorker的类的新实例。现在你就可以有一个包含多种不同类型对象的数组了！

如果试图实例化接口，编译器会报错。

**这是不允许的……**

```
IStingPatrol dennis = new IStingPatrol();
```

1 Cannot create an instance of the abstract class or interface

不能对一个接口使用new关键字，这是有道理的，接口中的方法和属性并没有具体实现。如果可以从一个接口创建对象，那它怎么知道要做什么呢?

**……不过这是允许的。**

```
NectarStinger fred = new NectarStinger();
IStingPatrol george = fred;
```

还记得吗？如果需要Sandwich类，总可以传入一个BLT引用取而代之，因为BLT继承自Sandwich。这里也是一样的。需要IStingPatrol的任何方法或语句中都可以使用NectarStinger引用。

尽管这个对象还可以做更多工作，不过使用接口引用时，就只能访问该接口中的方法。

第一行是一个正常的new语句，创建了名为fred的引用，并指向一个NectarStinger对象。

第二行有点意思了，因为这行代码使用**IStingPatrol**创建了一个新的引用变量。这一行乍看起来有点奇怪。不过再看下面的语句：

```
NectarStinger ginger = fred;
```

应该知道这个第三个语句做什么，它会创建一个新的NectarStinger引用，名为giner，并指向fred所指向的对象。声明george的那行语句也做了同样的工作，只不过使用了接口IStingPatrol。

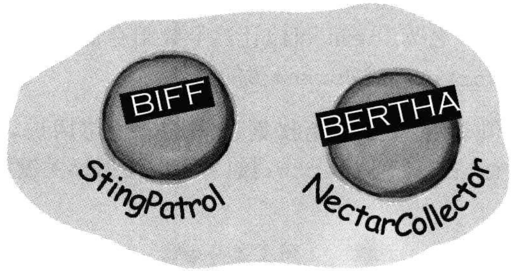



### 发生了什么?

这里只有一个new语句，所以只会创建一个对象。第二个语句创建了一个名为george的引用变量，它可以指向实现了IStingPatrol的任何类的一个实例。

# 接口引用类似于对象引用

你已经知道对象存放在堆中。使用接口引用时，实际上只是提供了另一个途径来指示所处理的对象。请看，这非常简单！

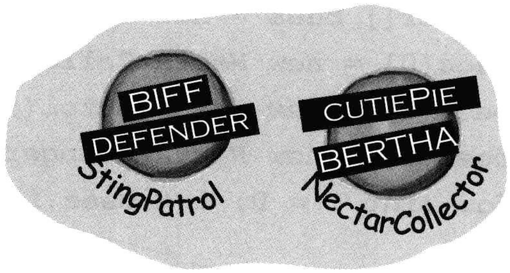



**1 创建两个蜜蜂对象。**

现在这对你来说应该很熟悉了。

```
StingPatrol biff = new StingPatrol();
NectarCollector bertha = new NectarCollector();
```

假设StingPatrol实现了IStingPatrol接口，而NectarCollector实现了INectarCollector接口。

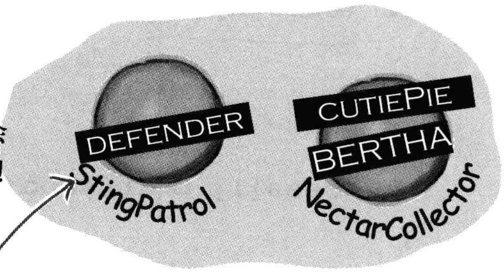



**2 增加IStingPatrol和INectarCollector引用。**

可以像使用其他引用类型一样使用接口引用。

```
IStingPatrol defender = biff;
INectarCollector cutiePie = bertha;
```

这两个语句使用接口来创建现有对象的新引用。只能将接口引用指向实现了该接口的某个类的实例。

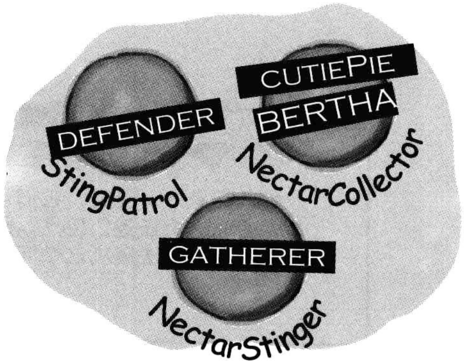



**3 接口引用可以保持一个对象存活。**

对于一个对象，如果没有任何引用指向它，则这个对象就会消失。但是并没有规定要求这些引用都必须有相同的类型！对于跟踪对象来说，接口引用与对象引用同样有效。

```
biff = null;
```

这个对象不会消失，因为defender还指向这个对象。




**4 将一个新实例赋至一个接口引用。**

可能并不真的需要一个对象引用，完全可以创建一个新对象，把它直接赋给一个接口引用变量。

```
INectarCollector gatherer = new NectarStinger();
```

# 可以使用"is"查看一个类是否实现了某个接口

有时需要得出某个类是否实现了一个接口。假设所有工蜂都存放在一个数组中，名为Bees。可以让这个数组存储Worker类型的对象，因为所有工蜂都是Worker类或Worker类的子类。

但是哪些工蜂能收集花露呢？换句话说，我们希望知道这个类是否实现了INectarCollector接口。可以使用is关键字找出答案。

所工工蜂都在一个Worker数组中。我们将使用"is"得出每个蜜蜂具体是何种类型的工蜂。

已经有一个工蜂数组，要求这些工蜂完成一个花露收集任务。因此循环处理这个数组，并使用"is"得出哪些对象有适当的方法和属性来完成这个任务。

```
Worker[] bees = new Worker[3];
bees[0] = new NectarCollector();
bees[1] = new StingPatrol();
bees[2] = new NectarStinger();
for (int i = 0; i < bees.Length; i++)
{
    if (bees[i] is INectarCollector)
    {
        bees[i].DoThisJob("Nectar Collector", 3);
    }
}
```

利用is可以比较接口，也可以比较其他类型！

这表示，如果这个蜜蜂实现了INectarCollector接口……则做下面的工作。

既然知道这个蜜蜂能收集花露，可以为它分配收集花露的任务。

## there are no Dumb Questions

**问：请等等。我在接口中放一个属性时，它看起来就像一个自动属性。这是不是说实现接口时只能使用自动属性？**

**答：** 不，绝对不是。接口中的属性确实看上去非常类似一个自动属性，这一点不假，比如下一页上IWorker接口中的Job和ShiftsLeft。不过它们绝对不是自动属性。可以像这样实现Job：

```
public Job { get;
private set; }
```

需要有这个private set，因为自动属性要求同时有一个set和一个get（即使它们是私有的）。不过也可以这样来实现：

```
public job { get {
return "Accountant"; } }
```

对此编译器也会乐于接受。也可以增加一个set存取方法，接口要求有一个get，但是它并没有说不能同时有一个set（如果使用一个自动属性来实现，你可以自己决定这个set是私有还是公共）。

## BRAIN POWER

如果有另外一些类，它们没有继承Worker，但是这些类确实实现了INectarCollector接口，那它也能完成这个任务！但是，既然它没有继承Worker，就不能把它与其他蜜蜂放在一个数组中。你能想出一个办法解决这个问题吗？创建一个数组，其中既能存放蜜蜂，也能存放这个新类。

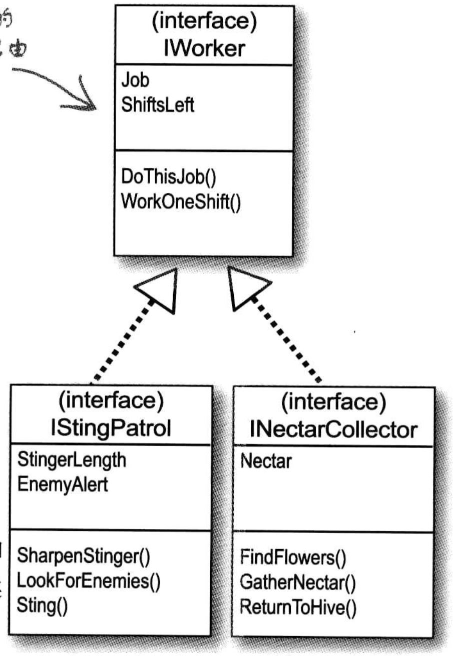
# 接口可以继承其他接口

一个类继承另一个类时，它会得到基类的所有方法和属性。接口的继承更加简单。由于所有接口中都没有具体的方法体，所以不需要考虑调用基类构造函数或方法的问题。继承接口（子接口）只需汇集它继承的接口的所有方法和属性。

**在类图中画接口时，会使用虚线显示接口的继承关系。**

```
interface IWorker
{
  string Job { get; }
  int ShiftsLeft { get; }
  void DoThisJob(string job, int shifts)
  void WorkOneShift()
}
```

我们创建了一个新的IWorker接口，可以由其他接口继承。




## 如果一个接口继承了IWorker，则实现该接口的任何类也必须实现IWorker的方法和属性

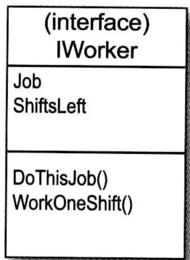
如果一个类实现了一个接口，则必须包含这个接口中的所有属性和方法。如果这个接口继承自另一个接口，那么后者的所有属性和方法也需要在这个类中实现。

```
interface IStingPatrol : IWorker
{
    int AlertLevel { get;}
    int StingerLength { get; set;}
    bool LookForEnemies();
    int SharpenStinger(int length);
}
```

还是原来的IStingPatrol接口，不过，现在它继承了IWorker接口。看上去只是一个很小的修改，不过对于实现IStingPatrol的类来说却有很大的差别。

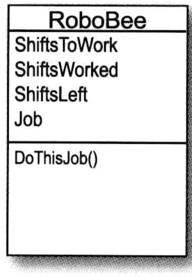
实现IStingPatrol的类不仅要实现这些方法……

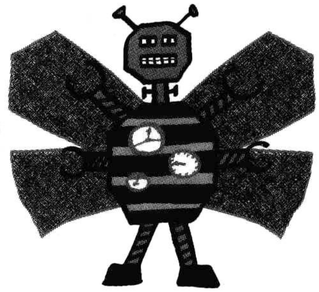
……还要实现这个接口继承的IWorker接口中的方法。



# RoboBee 4000可以完成工蜂的工作，而且不需要消耗珍贵的蜂蜜

下面创建一个新蜜蜂，RoboBee 4000，它不吃蜂蜜只消耗汽油。不过，可以让它继承IWorker接口，这样它就能像正常的工蜂一样工作了。



这是Robot基类，所以机器人要消耗汽油。

```
class Robot
{
     public void ConsumeGas() { ... }
}

class RoboBee : Robot, IWorker
{
     private int shiftsToWork;
     private int shiftsWorked;
     public int ShiftsLeft
       {get {return shiftsToWork - shiftsWorked;}}
     public string Job { get; private set; }
     public bool DoThisJob(string job, int shiftsToWork){ ... }
     public void WorkOneShift() { ... }
}
```

RoboBee类继承了Robot，并实现了IWorker。这说明，它是一个机器人，不过可以完成工蜂的工作！太棒了！

RoboBee类实现了IWorker接口中定义的所有方法。

如果RoboBee没有完全实现IWorker接口中的所有方法，这个代码将不能编译。

记住，对于应用中的其他类，RoboBee和正常的工蜂在功能上没有任何差别。它们都实现了IWorker接口，所以对程序中的其余部分来说它们都将看做是工蜂。

不过，可以使用is来区别这两种类型：

```
if (workerBee is Robot) {
  // now we know workerBee
  // is a Robot object
}
```

利用“is”可以看出workerBee实现了哪个接口或派生了哪个类。

**任何类可以实现任何接口，只要它能信守承诺，实现该接口的所有方法和属性。**

# is指出一个对象实现了什么，as则告诉编译器如何看待一个对象

有时需要调用对象的一个方法，而这个方法是由对象所实现的一个接口得到的。但是如果你不知道这个对象的类型是否正确又该怎么办呢？可以使用is来找出答案。明确对象的类型之后，可以使用as认为这个对象（你已经知道它有正确的类型）确实有你要调用的方法。

```
IWorker[] bees = new IWorker[3];
    bees[0] = new NectarStinger();
    bees[1] = new RoboBee();
    bees[2] = new Worker();
```

所有这些蜜蜂都实现了IWorker，但是我们不知道哪些蜜蜂还实现了其他接口，如INectarCollector。

循环处理各个蜜蜂……

不能对蜜蜂调用INectarCollector方法。它们的类型是IWorker，并不知道INectarCollector的方法。

```
for (int i = 0; i < bees.Length; i++) {
    if (bees[i] is INectarCollector) {
        INectarCollector thisCollector;
        thisCollector = bees[i] as INectarCollector;
        thisCollector.GatherNectar();
        ......
```

……并查看它是否实现了INectarCollector。

使用"as"就是说把这个对象看成是一个INectarCollector实现。

现在可以调用INectarCollector方法了。

## Sharpen your pencil

请观察左边的数组。对于以下每条语句，写出适当的i值使该语句计算为true。另外，左边有两个赋值语句不能编译，把它们画掉。

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

1. (Bees[i] is INectarCollector)

..............................................................

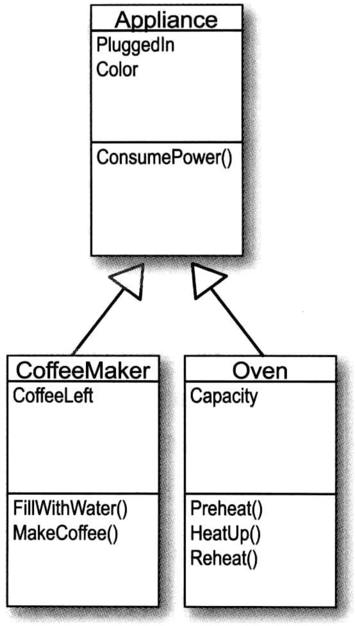
2. (Bees[i] is IStingPatrol)

..............................................................

3. (Bees[i] is IWorker)

..............................................................

# CoffeeMaker也是一个Appliance




如果你想知道怎么减少每个月的电费，可能不会关心每个家用电器做什么。你只关心它们的用电情况。所以，如果要写一个程序来监视用电情况，则可能只需写一个`Appliance`类。但是，如果需要区别咖啡机和电炉，就必须建立一个类层次体系。这样可以为`CoffeeMaker`和`Oven`类增加特定于咖啡机或电炉的方法和属性，而且这两个类都可以继承`Appliance`类，其中包含它们共有的一些方法和属性。

```
public void MonitorPower(Appliance appliance) {
    // code to add data to a household
    // power consumption database
}
```

程序使用这个方法监视一幢房子的用电情况。

这个代码在后面监视咖啡机用电情况的程序中还会出现。

```
CoffeeMaker misterCoffee = new CoffeeMaker();
MonitorPower(misterCoffee);
```

尽管MonitorPower()方法取一个Appliance对象引用作为参数，不过可以传入misterCoffee引用，因为CoffeeMaker是Appliance的一个子类。

在上一章已经见过这样的例子，可以把一个BLT引用传入需要Sandwich的方法。

## Sharpen your pencil Solution

请观察左边的数组。对于以下每条语句，写出适当的i值使该语句计算为true。另外，左边的两个赋值语句不能编译，把它们画掉。

```
IWorker[] Bees = new IWorker[8];
Bees[0] = new NectarStinger();
Bees[1] = new RoboBee();
Bees[2] = new Worker();
Bees[3] = Bees[0] as IWorker;
Bees[4] = IStingPatrol;
Bees[5] = null;
Bees[6] = Bees[0];
Bees[7] = new INectarCollector();
```

（Bees[4]和Bees[7]两行已被画掉。）

NectarStinger()实现了IStingPatrol接口。

1. (Bees[i] is INectarCollector)

   0和6

2. (Bees[i] is IStingPatrol)

   0, 6

3. (Bees[i] is IWorker)

   0, 1, 2, 3和6

# 对象和接口的向上强制转换

用子类代替基类时（比如用一个咖啡机替代家用电器，或者使用BLT取代三明治），这称为向上强制转换（upcasting）。在构建类层次体系时这确实是一个强大的工具。向上强制转换唯一的缺点是，只能使用基类的属性和方法。换句话说，把一个咖啡机看成是一个家用电器时，就不能告诉它煮咖啡或加满水。不过可以告诉它是否插上插头，因为所有家用电器都能做这个工作（正因如此，`PluggedIn`属性属于`Appliance`类）。

**1 下面创建一些对象。**

可以正常地创建`CoffeeMaker`和`Oven`类：

```
CoffeeMaker misterCoffee = new CoffeeMaker();
Oven oldToasty = new Oven();
```

首先正常地实例化一个Oven对象和一个CoffeeMaker对象。

**2 如果想创建一个家用电器数组呢？**

不能把`CoffeeMaker`放在一个`Oven[ ]`数组中，也不能把`Oven`放在一个`CoffeeMaker[ ]`数组中。不过这两种对象都可以放在`Appliance[ ]`数组中：

```
Appliance[] kitchenWare = new Appliance[2];
kitchenWare[0] = misterCoffee;
kitchenWare[1] = oldToasty;
```

可以使用向上强制转换创建一个家用电器数组，咖啡机和电炉都可以放在这个数组中。

**3 但是不能把家用电器看做是电炉。**

得到一个`Appliance`引用时，只能访问与家用电器有关的方法和属性。不能通过`Appliance`引用使用咖啡机特定的方法和属性（即使你已经知道这确实是一个`CoffeeMaker`）。所以以下语句能正常工作，因为它们把一个`CoffeeMaker`对象看做是一个`Appliance`：

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

powerConsumer是一个指向CoffeeMaker对象的Appliance引用。

但是一旦试图把它用做一个咖啡机（CoffeeMaker）：

```
powerConsumer.MakeCoffee();
```

这一行不能编译，因为powerConsumer是一个Appliance引用，所以只能用来完成Appliance的工作。

代码将不能编译，IDE会显示这样一个错误：

```
'Appliance' does not contain a
definition for 'MakeCoffee'
```




这是因为，一旦将一个子类向上强制转换为基类，由于访问对象使用的是基类引用，所以只能访问该基类的方法和属性。

# 向下强制转换允许你把家用电器变回咖啡机

向上强制转换是一个很不错的工具，因为利用向上强制转换，任何地方需要一个家用电器时，完全可以使用一个咖啡机或电炉。但是，这里有一个很严重的缺点，如果使用一个指向CoffeeMaker对象的Appliance引用，就只能使用Appliance的方法和属性。这就引入了向下强制转换（downcasting）：也就是将先前的向上强制转换引用变回来。可以使用is关键字明确一个Appliance对象是否实际上是一个CoffeeMaker。一旦确定，可以使用as关键字把这个Appliance再变回为一个CoffeeMaker。

这就是上一页的Appliance引用，它指向一个CoffeeMaker对象。




**1 先从前面向上强制转换的咖啡机开始。**

以下是前面使用的代码：

```
Appliance powerConsumer = new CoffeeMaker();
powerConsumer.ConsumePower();
```

**2 但是如果想把这个Appliance变回为CoffeeMaker呢？**

向下强制转换的第一步是使用is关键字检查有没有这种可能。

```
if (powerConsumer is CoffeeMaker)
    // then we can downcast!
```

**3 既然知道这确实是一个CoffeeMaker，下面就把它用做一个咖啡机。**

is关键字只是第一步。一旦知道这是一个指向CoffeeMaker对象的Appliance引用，就可以使用as关键字完成向下强制转换。这样就可以使用CoffeeMaker类的方法和属性了。另外，由于CoffeeMaker继承自Appliance，所以仍然享有Appliance的方法和属性。

```
if (powerConsumer is CoffeeMaker) {
    CoffeeMaker javaJoe = powerConsumer as CoffeeMaker;
    javaJoe.MakeCoffee();
}
```

javaJoe引用指向powerConsumer所指向的同一个CoffeeMaker对象。不过这是一个CoffeeMaker引用，所以可以调用MakeCoffee()方法。



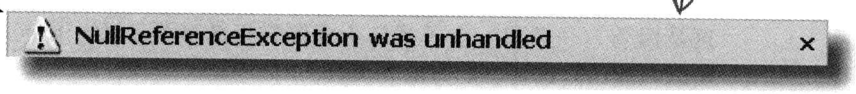

## 向下强制转换失败时，as返回null

如果想使用as把一个Oven对象转换为CoffeeMaker会怎么样呢？它会返回null，倘若试图使用这个对象，.NET将使程序崩溃。

```
if (powerConsumer is CoffeeMaker) {
    Oven foodWarmer = powerConsumer as Oven;
    foodWarmer.Preheat();
}
```

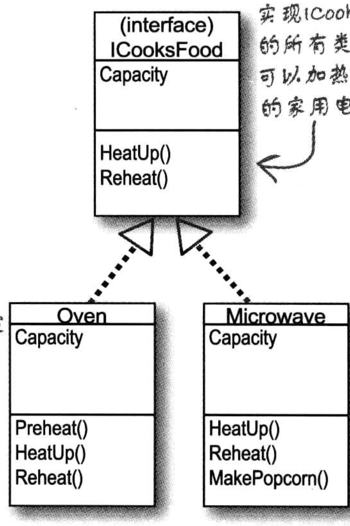
唉呀，这两个类型不匹配！

powerConsumer不是一个Oven对象。所以想用"as"向下强制转换时，foodWarmer引用最后会设置为null。而试图使用一个null引用就会导致这种结果……



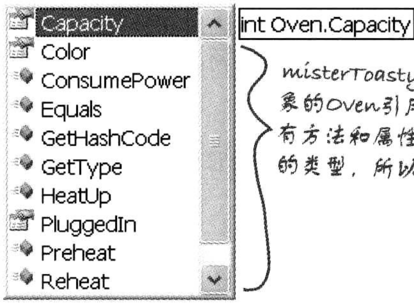
# 接口也可以向上和向下强制转换

你已经知道可以对接口使用is和as。下面就来尝试所有这些强制转换。先为所有能加热食物的类增加一个ICooksFood接口。再增加一个Microwave类，Microwave和Oven都实现了ICooksFood接口。现在访问Oven对象就有3种办法。IDE的智能提示可以帮助你明确使用这些办法时能做什么以及不能做什么：




实现ICooksFood的所有类都是可以加热食物的家用电器。

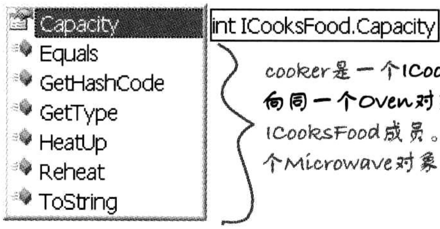
```
Oven misterToasty = new Oven();
misterToasty.
```




一旦输入点号，就会弹出智能提示窗口，显示所有可用成员的一个列表。

misterToasty是一个指向Oven对象的Oven引用，所以可以访问所有方法和属性……但是这是最特定的类型，所以只能指向Oven对象。

```
ICooksFood cooker;
if (misterToasty is ICooksFood)
    cooker = misterToasty as ICooksFood;
    cooker.
```

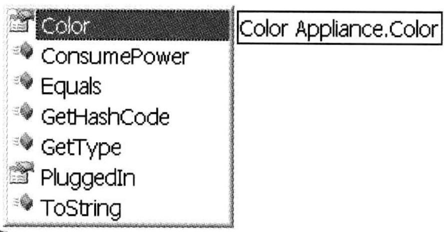



cooker是一个ICooksFood引用，指向同一个Oven对象。它只能访问ICooksFood成员。不过它还可以指向一个Microwave对象。

**这3个不同的引用都指向同一个对象，取决于引用的类型，它们能访问的方法和属性是不同的。**

```
Appliance powerConsumer;
if (misterToasty is Appliance)
    powerConsumer = misterToasty;
    powerConsumer.
```

powerConsumer是一个Appliance引用。通过它只能访问Appliance中的公共字段、方法和属性。如果愿意，还可以用它指向一个CoffeeMaker对象。



## there are no Dumb Questions

**问：等一等，你说过，总是可以向上强制转换，但不一定总能向下强制转换，为什么呢？**

**答：**因为如果向上强制转换失败时编译器会警告你。只有一种情况下向上强制转换无法进行，这就是试图将一个对象设置为某个类而它并没有继承这个类，或者将它设置为某个接口而它没有实现这个接口。编译器会立即发现你的向上强制转换不合适，并给出一个错误。

但另一方面，编译器不知道怎么检查向下强制转换，如果你把一个对象或接口引用向下强制转换为一个不合法的引用，编译器将无从得知。这是因为，把任何类或接口名放在as关键字的右边都是完全合法的。如果向下强制转换是非法的，as语句只会返回null。编译器不会阻止你这样做，这可能是一件好事，因为很多情况下你确实想这么做。

**问：有人告诉我说，接口就像一个契约，不过我不清楚为什么。这是什么意思？**

**答：**对，我们也听说过，很多人都喜欢说接口就像一个契约（面试时这确实是一个经常问到的问题）。在某种程度上，确实如此。让类实现一个接口时，就是在告诉编译器你承诺在类中放某些方法。编译器会保证你信守这个承诺。

但是我们认为，要想记住接口是如何工作的，把接口看成是一种名单可能更为容易。编译器会检查这个名单，确保你确实把接口的所有方法都已经放在类中。如果没有做到，编译器就会中止，不允许编译。

**问：如果我想把一个方法体放在接口中会怎么样呢？这样行吗？**

**答：**不行，编译器不允许你这样做。接口中不允许有任何语句。尽管实现接口同样使用冒号操作符，但这与继承类完全是两码事。实现一个接口不会为类增加任何行为，也不会对类做任何改变。实现接口时，只是告诉编译器要确保类中有该接口定义的所有方法（也就是要求这个类必须有所有这些方法）。

**问：那何必使用接口呢？看起来它只是增加限制，根本没有改变类。**

**答：**这是因为，如果一个类实现了一个接口，接口引用就可以指向这个类的任何实例。这对你来说非常有用，你可以创建一个引用类型来处理多种不同类型的对象。

下面是一个简单的例子。马、牛、骡子和食用牛都能拉车。但是在我们的动物园模拟系统中，Horse、Ox、Mule和Steer分别有不同的类。假设动物园里有一个拉车游行，你希望创建一个数组，其中可以包含能拉车的任何动物。很糟糕，你不能创建这样一个能容纳所有拉车动物的数组。如果它们都继承同一个基类，那么创建这样一个数组是可以的。但是它们并没有继承同一个基类。这该如何是好呢？

此时接口就有用武之地了。可以创建一个IPuller接口，其中包含拉车的有关方法。可以如下声明数组：

```
IPuller[] pullerArray;
```

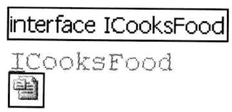
现在任何动物只要实现了IPuller接口，就可以把它的引用放在这个数组中。

**问：有没有更容易的方法来实现接口？要输的内容太多了！**

**答：**没错，有个好办法。IDE提供了一个功能很强大的简便方法，它会自动为你实现一个接口。首先如下输入一个类：

```
public class
  Microwave : ICooksFood
  { }
```

单击ICooksFood，会看到"I"下方出现一个小条，把鼠标停在这个小条上，你会看到它下面出现一个图标：



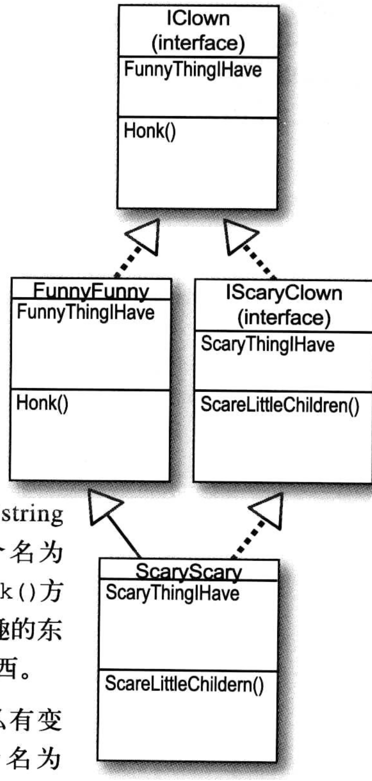

有时很难点到这个图标，不过按Ctrl加点号也能达到目的。

单击这个图标，并从菜单选择"Implement Interface 'ICooksFood'"。它会自动增加这个接口中尚未实现的所有方法。每个方法中只有一个throws语句。这会导致程序中止，以此提醒你不要忘记实现这些方法（第10章将更多地了解throws）。

**接口就像一个名单，编译器检查这个名单来确保类实现了指定的一组方法。**

**Exercise**

扩展IClown接口，并使用实现这个接口的类。




❶ 先从277页上最后一个“动手做！”例子中的IClown接口开始：

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}
```

❷ 创建一个新接口IScaryClown扩展IClown，这个新接口继承了IClown。它另外有一个string属性，名为ScaryThingIHave，这个属性有一个获取存取方法但没有设置存取方法。此外还有一个void方法ScareLittleChildren()。

❸ 创建以下类：

★ 一个有趣小丑类，名为FunnyFunny，它使用一个私有的string变量存储一个有趣的东西，并使用构造函数（有一个名为funnyThingIHave的参数）来设置这个私有字段。Honk()方法应当显示“Honk honk! I have a ”，后面是那个有趣的东西。FunnyThingIHave设置存取方法应当返回同一个东西。

★ 一个可怕小丑类，名为ScaryScary，它使用一个私有变量存储一个整数，这个整数由其构造函数通过一个名为numberOfScaryThings的参数传入。ScaryThingIHave获取存取方法要返回一个串，包括构造函数传入的这个数，后面是“spiders”。ScareLittleChildren()弹出一个消息框，显示“Boo! Gotcha!”。

❹ 以下是一个按钮的代码，但是它不能正常工作。你能找出问题并解决这个问题吗？

```
private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown;
    someOtherScaryclown.Honk();
}
```

惹恼小丑可不是开玩笑的。



## Exercise Solution

扩展IClown接口，并使用实现这个接口的类。

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
string ScaryThingIHave { get; }
    void ScareLittleChildren();
}

class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    private string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Honk honk! I have " + funnyThingIHave; }
    }

    public void Honk() {
        MessageBox.Show(this.FunnyThingIHave);
    }
}

class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave, int numberOfScaryThings)
        : base(funnyThingIHave) {
        this.numberOfScaryThings = numberOfScaryThings;
    }

    private int numberOfScaryThings;
    public string ScaryThingIHave {
        get { return "I have " + numberOfScaryThings + " spiders"; }
    }

    public void ScareLittleChildren() {
        MessageBox.Show("Boo! Gotcha!");
    }
}
private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown as ScaryScary;
    someOtherScaryclown.Honk();
}
```

Honk()方法使用这个设置存取方法显示消息，不需要把这个代码重复两次。

也可以再实现IClown方法和属性，不过为什么不直接从FunnyFunny继承呢？

由于ScaryScary是FunnyFunny的一个子类，而FunnyFunny实现了IClown，所以ScaryScary也实现了IClown。

可以把一个FunnyFunny引用设置为等于一个ScaryScary对象，因为ScaryScary继承自FunnyFunny。不过，不能把IScaryClown引用设置为任何小丑，因为你不知道这个小丑是不是可怕。正是因为这个原因，所以需要as关键字。

也可以使用someOtherScaryClown引用来调用ScareLittleChildren()，但是不能从someFunnyClown引用调用这个方法。

# 不只是公共和私有

你已经知道private关键字的重要性，也知道如何使用这个关键字，以及它与public有什么区别。C#对这些关键字有一个专门的名字，称之为访问修饰符（access modifiers）。这个名字很有道理，因为改变类的一个成员（属性、字段或方法）或整个类的访问修饰符时，也就改变了其他类将如何访问这个类或成员。还可以使用另外一些访问修饰符，不过先来看你已经了解的访问修饰符：

我们称一个类的方法、字段和属性为它的成员（members）。所有成员都可以标志public或private访问修饰符。

（只能它们可以访问声明类）

- **public表示任何人都可以访问。**
  将一个类或类成员标志为public时，就是在告诉C#：其他类的任何实例都允许访问。这是限制最弱的访问修饰符。你已经看到，这个访问修饰符可能会带来麻烦，只有当有充分的理由时才可以把类成员标志为公共。这样才能确保类得到很好的封装。

- **private表示只有其他成员可以访问。**
  将一个类成员标志为private时，只能从该类中的成员或该类其他实例来访问。不能把一个类标志为private，除非这个类位于另一个类内部，在这种情况下，它只对其"容器"类的实例可用。类成员默认为私有，如果希望它是公共的，则需要明确标识为公共。

声明一个类成员时如果没有访问修饰符，则默认为private。

- **protected对于子类表示public，对其他则表示private。**
  你已经知道子类不能访问其基类中的私有字段，另外必须使用base关键字才能访问基类对象的公共成员。如果子类能访问这些私有字段则应该会很方便。正因如此，引入了protected访问修饰符。标志protected的类成员可以由该类中所有其他成员访问，另外该类子类中的所有成员也可以访问。

- **internal只对同一个程序集中的其他类表示public。**
  内置.NET Framework类是一些程序集，也就是工程引用列表中的类库。在Solution Explorer中鼠标右键单击"References"，并选择"Add Reference……"可以看到一个程序集列表，创建一个新的Windows Forms Application时，IDE会自动包含构建Windows应用所需的引用。构建一个程序集时，可以使用internal关键字使类对该程序集私有，从而可以只公开你希望公开的类。这个关键字可以与protected一同使用，标识为protected internal的所有内容都只能在程序集内部或从一个子类访问。

声明一个类或接口时如果没有访问修饰符，则默认设置为internal。对于大多数类来说这是可以的，这说明程序集中的所有其他类都可以访问这个类。如果没有使用多个程序集，internal对于类和接口来说就相当于public。可以试一试，打开一个原先的工程，将一些类的访问修饰符改为internal，看看会发生什么。

- **sealed 表示这个类不能派生子类。**
  有一些类不允许其他类继承。很多.NET Framework类就是如此，可以试一试，建立一个继承String的类（上一章使用了String类的IsEmptyOrNull()方法），会发生什么情况？编译器不允许构建这个代码，它会给出错误"cannot derive from sealed type 'string'"（无法派生密封类型'string'）。也可以让你自己的类做到这一点，只需在访问修饰符后面增加sealed。

sealed是一个修饰符，而不是访问修饰符。这是因为它只影响继承性，并不改变访问类的方式。

**对以上这些定义还有一些说明。可以先看看附录"其他"中的第2项，了解更多有关内容。**

# 访问修饰符改变可见性

下面仔细分析这些访问修饰符，了解它们对各个类成员的作用域有什么影响。我们做了两处修改：funnyThingIHave后备字段现在是protected，另外修改了ScareLittleChildren()方法，它要使用这个funnyThingIHave字段。

对你自己的练习答案做这两个修改。然后把protected访问修饰符再改回为private，看看会得到什么错误。

❶ 这里有两个接口。IClown定义了一个吹喇叭而且有一个有趣玩艺的小丑。IScaryClown继承小丑（IClown）。可怕小丑可以做小丑做的所有事情，另外他还有一个可怕玩艺，可以吓小孩子。

```
interface IClown {
    string FunnyThingIHave { get; }
    void Honk();
}

interface IScaryClown : IClown {
    string ScaryThingIHave { get; }
    void ScareLittleChildren();
}
```

"this"也会改变指示的是哪一个变量。它告诉C#，"查看类的当前实例，找出我连接的字段，即使它与一个参数或局部变量同名"。

这是"this"的一种常见用法，因为参数和后备字段同名。funnyThingIHave指示参数，而this.funnyThingIHave是后备字段。

❷ FunnyFunny类实现了IClown接口。我们设置funnyThingIHave字段为保护字段（protected），从而可以由FunnyFunny子类的任何实例访问。

```
class FunnyFunny : IClown {
    public FunnyFunny(string funnyThingIHave) {
        this.funnyThingIHave = funnyThingIHave;
    }
    protected string funnyThingIHave;
    public string FunnyThingIHave {
        get { return "Honk honk! I have " + funnyThingIHave; }
    }

    public void Honk() {
        MessageBox.Show(this.FunnyThingIHave);
    }
}
```

通过增加"this"，告诉C#我们所说的是后备字段，而不是同名的参数。

把FunnyThingIHave改为protected，看看它对ScaryScary.ScareLittleChildren()方法有什么影响。

在属性前使用"this"时，就是在告诉C#要执行设置或获取存取方法。

## 访问修饰符放大镜

**3** ScaryScary类实现了IScaryClown接口。它还继承了FunnyFunny，由于FunnyFunny实现了IClown，这说明ScaryScary也实现了IClown。下面来看ScareLittleChildren()方法如何访问funnyThingIHave后备字段，它可以访问这个字段是因为我们使用了protected访问修饰符。如果把它设置为私有字段，则这个代码就无法编译了。

```
class ScaryScary : FunnyFunny, IScaryClown {
    public ScaryScary(string funnyThingIHave,
                             int numberOfScaryThings)
         : base(funnyThingIHave)  {
         this.numberOfScaryThings = numberOfScaryThings;
    }

    private int numberOfScaryThings;
    public string ScaryThingIHave {
         get { return "I have " + numberOfScaryThings + " spiders"; }
    }

    public void ScareLittleChildren()  {
         MessageBox.Show("You can't have my "
                          + base.funnyThingIHave);
    }
}
```

numberOfScaryThings是私有的，这是一种典型的后备字段。所以只有ScaryScary的实例能看到它。

protected关键字告诉C#，除了子类的实例外，这个属性对其他对象都是私有的。

"base"关键字告诉C#使用基类中的值。但是在这里也可以使用"this"。你知道这是为什么吗？

如果funnyThingIHave仍保持为private，则会导致编译器报错。不过如果把它改为protected，就会对FunnyFunny的子类可见。

**4** 这个按钮会实例化FunnyFunny和ScaryScary。来看它如何使用as将someFunnyClown向下强制转换为一个IScaryClown引用。

```
private void button1_Click(object sender, EventArgs e) {
    ScaryScary fingersTheClown = new ScaryScary("big shoes", 14);
    FunnyFunny someFunnyClown = fingersTheClown;
    IScaryClown someOtherScaryclown = someFunnyClown as ScaryScary;
    someOtherScaryclown.Honk();
}
```

我们多加了几个步骤来展示可以将ScaryScary向上强制转换为FunnyFunny，然后再向下强制转换为IScaryClown。不过这三行代码可以合并为一行代码。你知道怎么做吗？

由于这个按钮的click事件处理程序不是FunnyFunny和ScaryScary的一部分，所以不能访问protected的funnyThingIHave字段。

它在两个类外部，所以其中的语句只能访问FunnyFunny或ScaryScary对象的公共成员。

## there are no Dumb Questions

**问：为什么要使用接口，而不是在类中直接编写我需要的所有方法？**

**答：**在编写越来越复杂的程序时，会有大量不同的类。利用接口，可以按这些类完成的工作对它们分组。接口还可以帮助你确保完成某个工作的各个类在使用同样的方法。类可以完成需要它处理的任何工作，而且，基于接口，你不需要操心它具体是怎样做的，只要能完成任务就行。

下面是一个例子：可能有一个卡车类和一个帆船类都实现了ICarryPassenger。假设ICarryPassenger接口要求实现这个接口的所有类都必须有一个ConsumeEnergy()方法。这两个类都可以用来载客，不过帆船类的ConsumeEnergy()方法使用风力，而卡车类的这个方法使用柴油。

假设没有ICarryPassenger接口。要想告诉程序哪一个能载客而哪一个不能，这会很困难。必须检查程序可能使用的每一个类，确定其中有没有一个方法可以将人从一处载到另一处。然后必须调用程序中将使用的各种交通工具（其中包含定义的某个载客方法）。由于没有标准接口，这些方法的名字可能花样百出，甚至与其他方法冲突。很快你就会看到这会造成怎样的混乱。

**问：为什么需要使用属性呢？不能直接包括一个字段吗？**

**答：**这个问题问得很好。接口只能定义一个类以何种方式完成一个特定的任务。它本身不是对象，所以不能实例化接口，它也不能存储信息。如果增加一个字段（就是一个变量声明），则C#就必须在某个地方存储这个数据，但接口本身不能存储数据。利用属性，可以让一个东西对于其他对象来说看起来像是一个字段，但由于它实际上是一个方法，这样就不会真正存储任何数据。

**问：常规的对象引用和接口引用之间有什么区别？**

**答：**你已经知道常规的对象引用是如何工作的。如果创建一个名为VertBoard的Skateboard实例，则再创建一个指向这个对象的新引用，名为HalfPipeBoard，它们都指向同一个东西。但是如果Skateboard实现了接口IStreetTricks，而你创建了一个指向Skateboard的接口引用，名为StreetBoard。它就只知道Skateboard类中IStreetTricks接口所定义的方法。

所有这3个引用实际上都指向同一个对象。如果使用HalfPipeBoard或VertBoard引用调用对象，就能访问这个对象中的任何方法或属性。如果使用StreetBoard引用来调用，就只能访问IStreetTricks接口中的方法和属性。

**问：既然接口引用会限制对象的作用，为什么还要使用接口呢？**

**答：**接口引用提供了一种途径，可以处理完成同一件事的多种不同类型的对象。可以使用接口引用类型创建一个数组，这样一来，不论处理的是一个卡车、马、独轮车还是小汽车对象，都可以为ICarryPassenger中的方法传递信息以及获得数据。这些对象完成这个任务的具体做法可能稍有差异，但是基于接口引用，可以知道它们都有同样的方法（取相同的参数，并有相同的返回类型）。所以，可以用完全相同的方式调用这些方法以及传递信息。

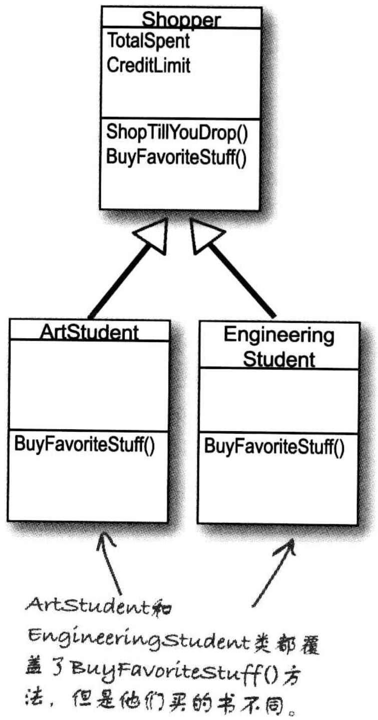
**问：为什么要声明为protected而不是private或public？**

**答：**因为这可以更好地封装你的类。很多情况下，子类需要访问其基类内部的某个部分。例如，如果需要覆盖一个属性，可以在获取存取方法中使用基类的后备字段，然后对基类的后备字段有所调整后返回，这种做法很常见。但是构建类时，只有在有充分理由的情况下才能声明为公共成员。使用protected访问修饰符可以把这一部分只对需要它的子类公开，而对所有其他类都是私有的。

**接口引用只知道该接口中定义的方法和属性。**

# 有些类不应实例化

还记得我们的动物园模拟系统类层次体系吗？最后你肯定会实例化很多河马、狗和狮子对象。但是Canine和Feline类呢？Animal类呢？看起来，有些类不需要实例化…… 而且，实际上，对这些类实例化没有任何意义。请看下面的例子。




先看一个基本类，表示在学生书店买书的一个学生。

```
class Shopper {
    public void ShopTillYouDrop()
        while (TotalSpent < CreditLimit)
            BuyFavoriteStuff();
    }
    public virtual void BuyFavoriteStuff () {
        // No implementation here - we don't know
        // what our student likes to buy!
    }
}
```

这是ArtStudent类，它派生了Shopper:

```
class ArtStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyArtSupplies();
        BuyBlackTurtlenecks();
        BuyDepressingMusic();
    }
}
```

EngineeringStudent类也继承了Shopper:

```
class EngineeringStudent : Shopper {
    public override void BuyFavoriteStuff () {
        BuyPencils();
        BuyGraphingCalculator();
        BuyPocketProtector();
    }
}
```

**实例化Shopper会怎么样呢？这样做有意义吗？**

# 抽象类就像类和接口之间的一个过渡

假设需要一个“接口”，要求类实现某些方法和属性。但是需要在这个“接口”中包含一些代码，从而不必在每个派生类中重复地实现某些方法。对于这种情况，你需要的并不是接口，而是一个抽象类（abstract class）。使用抽象类，可以获得接口的一些特性，不过还可以像正常的类一样在其中编写代码。

**抽象类就像一个正常的类。**
抽象类的定义类似于正常的类。抽象类可以有字段和方法，而且也可以继承其他类，这与正常类非常相似。这方面没有什么新东西需要学习，因为你已经知道抽象类能做什么！

如果一个方法有声明但是没有语句或方法体，这称为一个抽象方法（abstract method）。就像继承接口一样，继承抽象类的子类必须实现所有抽象方法。

只有抽象类能有抽象方法。如果把一个抽象方法放在一个类中，就必须标志这个类为抽象类，否则无法编译。稍后将更多地了解如何将一个类标志为abstract。

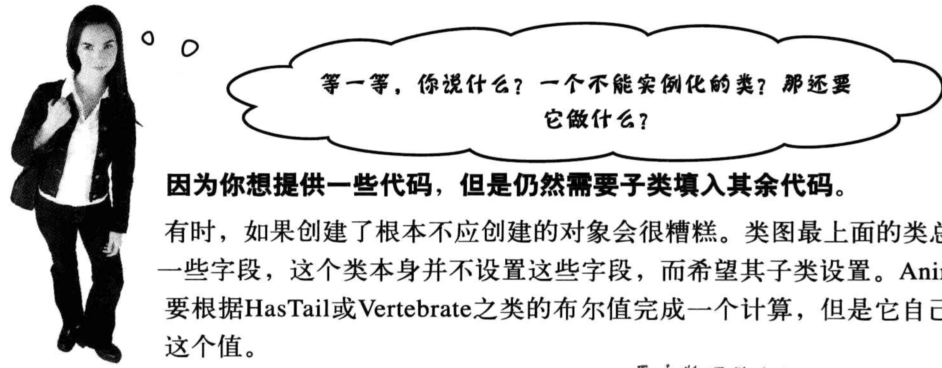
**抽象类就像一个接口。**
创建了一个实现了某个接口的类时，就说明已经同意实现这个接口中定义的所有属性和方法。抽象类也是如此，其中可以包含属性和方法的声明，与接口一样，这些属性和方法必须由继承类实现。

抽象的反义词是具体。具体方法是指有方法体的方法，目前为止处理的所有类都是具体类。

**不过抽象类不能实例化。**
抽象类和具体类之间最大的区别在于，不能使用new创建抽象类的实例。如果这样做，编译代码时C#将给出一个错误。

```
Cannot create an instance of the
abstract class or interface 'MyClass'
```

产生这个错误的原因是有一些抽象方法但没有相应的代码！编译器不允许实例化接口，类似地，也不允许实例化缺少代码的类。




**因为你想提供一些代码，但是仍然需要子类填入其余代码。**

有时，如果创建了根本不应创建的对象会很糟糕。类图最上面的类总会有这样一些字段，这个类本身并不设置这些字段，而希望其子类设置。Animal类可能要根据HasTail或Vertebrate之类的布尔值完成一个计算，但是它自己无法设置这个值。

请看下面的例子……

这是Objectville Astrophysics Club向其他星球发射火箭所用的类。

```
class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  public int TimeNeeded() {
    return MilesToPlanet / (int) RocketSpeedMPH;
  }

  public string FuelNeeded() {
    return "You'll need "
      + MilesToPlanet * RocketFuelPerMile
      + " units of fuel to get there. It'll take "
      + TimeNeeded() + " hours.";
  }
}
```

在基类中设置这些字段没有意义，因为我们不知道要用哪个火箭或发射到哪个星球。

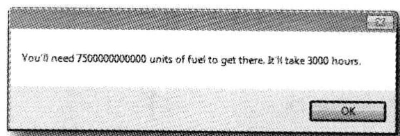
天文物理学家指定了两个任务，一个是发射到火星，另一个是发射到金星。

```
class Venus : PlanetMission {
    public Venus() {
        MilesToPlanet = 40000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}

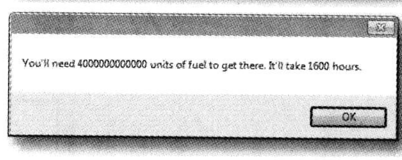
class Mars : PlanetMission {
    public Mars() {
        MilesToPlanet = 75000000;
        RocketFuelPerMile = 100000;
        RocketSpeedMPH = 25000;
    }
}
```

Mars和Venus子类的构造函数设置了从Planet继承的这3个字段。但是如果直接实例化Planet，这些字段不会设置。那么倘若FuelNeeded()试图使用这些字段会有什么结果？

```
private void button1_Click(object s, EventArgs e) {
  Mars mars = new Mars();
  MessageBox.Show(mars.FuelNeeded());
}
```




```
private void button2_Click(object s, EventArgs e) {
  Venus venus = new Venus();
  MessageBox.Show(venus.FuelNeeded());
}
```




```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

**翻到下一页之前，自己想想看用户单击第3个按钮时会有什么结果……**

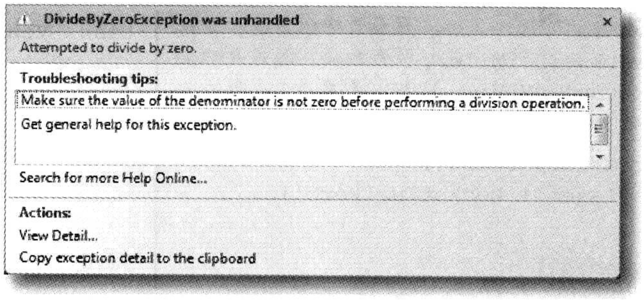
# 正像我们说的，有些类不应实例化

所有问题都是从创建PlanetMission类的实例引发的。这个类的FuelNeeded()方法希望子类设置这些字段。但是如果没有设置，就会得到默认值0。而当C#试图将一个数除以0时，就会……

```
private void button3_Click(object s, EventArgs e) {
  PlanetMission planet = new PlanetMission();
  MessageBox.Show(planet.FuelNeeded());
}
```

PlanetMission类本来就不应实例化。我们只打算继承这个类。但是一旦实例化这个类，问题就来了。

FuelNeeded()方法试图除以RocketSpeedMPH时，RocketSpeedMPH的值是0。除以0时就会导致这个错误。




## 解决方案:使用抽象类

将一个类标志为abstract时，C#不允许编写代码来实例化这个类。这很像接口，抽象类要作为其子类的一个模板。

为类声明增加abstract关键字，这就告诉C#这是一个抽象类，不能实例化。

```
abstract class PlanetMission {
  public long RocketFuelPerMile;
  public long RocketSpeedMPH;
  public int MilesToPlanet;

  public long UnitsOfFuelNeeded() {
    return MilesToPlanet * RocketFuelPerMile;
  }

  // the rest of the class is defined here
}
```

现在C#拒绝编译这个程序，除非把创建PlanetMission实例的那一行删除。

**BRAIN POWER**

翻回到上一章254～256页Kathleen宴会规划程序的解决方案，再来看那个代码里存在的封装问题。你能想出如何使用一个抽象类来解决那些问题吗？

# 抽象方法没有方法体

你已经知道接口只有方法和属性的声明，而实际上没有任何方法体，对不对。这是因为接口中的各个方法都是一个抽象方法。所以下面就来实现这些方法！一旦实现了方法，错误就会消失。扩展一个抽象类时，需要确保覆盖其所有抽象方法。幸运的是，IDE让这个工作变得更为容易。只需要输入“public override”。一旦按下空格键，IDE就会显示一个下拉框，其中包括可以覆盖的所有方法的一个列表。选择SetMissionInfo()方法，并在其中填入实现代码：

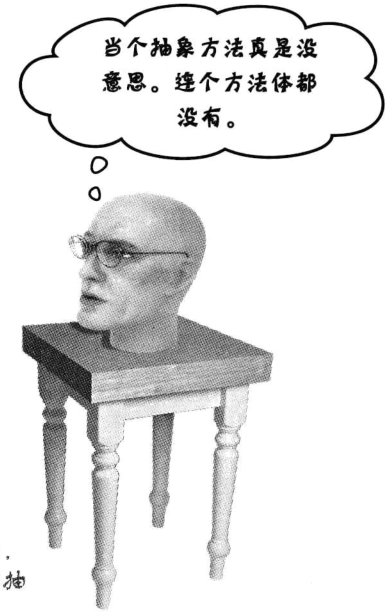
**接口中的各个方法会自动成为抽象方法，所以在接口中不需要像抽象类中那样使用abstract关键字。只有抽象类可以有抽象方法…… ，不过抽象类也可以有具体方法。**

```
abstract class PlanetMission {

    public abstract void SetMissionInfo(
        int milesToPlanet, int rocketFuelPerMile,
        long rocketSpeedMPH);

    // the rest of the class……
```

这个抽象方法就像接口中的方法，没有方法体，但是继承PlanetMission的所有类都必须实现SetMissionInfo()方法，否则程序不能编译。




如果加入这个方法，试图构建这个程序时，IDE会给出一个错误：

```
'VenusMission' does not implement inherited abstract
member 'PlanetMission.SetMissionInfo(long, int, int)'
```

所以下面来实现这个方法!一旦实现了方法，错误就会消失。

```
class Venus : PlanetMission {
    public Venus() {
        SetMissinInfo(40000000, 100000, 25000);
    }
    public override SetMissionInfo(int milesToPlanet, long rocketFuelPerMile,
     int rocketSpeedMPH) {
        this.MilesToPlanet = milesToPlanet;
        this.RocketFuelPerMile = rocketFuelPerMile;
        this.RocketSpeedMPH = rocketSpeedMPH;
    }
}
```

继承一个抽象类时，需要覆盖它的所有抽象方法。

## Sharpen your pencil

现在给你一个机会来展示你的艺术才能。左边是一组类和接口声明，你的任务是在右边画出相关的类图。我们已经为你完成了第一题。不要忘记用虚线表示实现一个接口，用实线表示继承一个类。

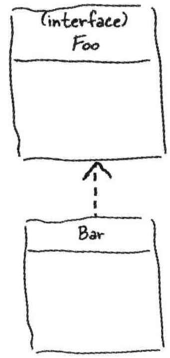
**给定：**

**类图？**

**1)**

```
interface Foo { }
class Bar : Foo { }
```

**1)**




**2)**

```
interface Vinn { }
abstract class Vout : Vinn { }
```

**2)**

**3)**

```
abstract class Muffie : Whuffie { }
class Fluffie : Muffie { }
interface Whuffie { }
```

**3)**

**4)**

```
class Zoop { }
class Boop : Zoop { }
class Goop : Boop { }
```

**4)**

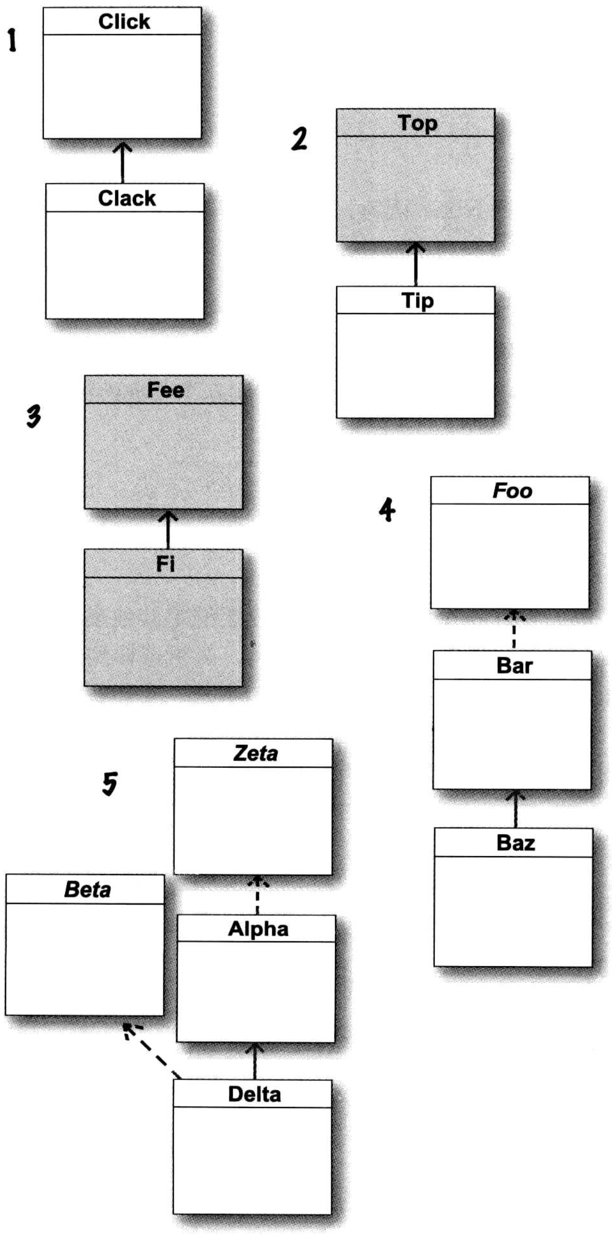
**5)**

```
class Gamma : Delta, Epsilon { }
interface Epsilon { }
interface Beta { }
class Alpha : Gamma,Beta { }
class Delta { }
```

**5)**

左边是一组类图。你的任务是把这些类图转换为合法的C#声明。我们已经为你完成了第一题。

**给定:**




**声明?**

1) public class Click { }

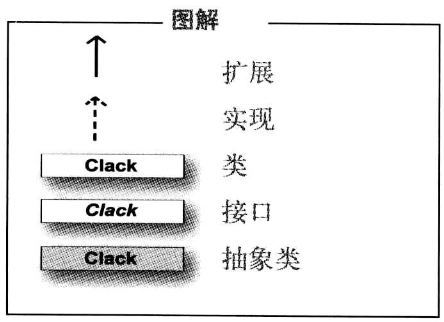
public class Clack : Click { }

2)

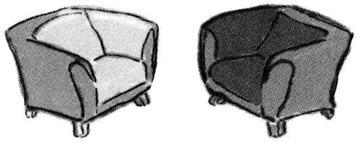
3)

4)

5)



# Fireside Chats



**今晚话题：一个抽象类和一个接口正在针锋相对地争论一个紧急问题，“谁更重要？”**

**抽象类：**

我想我们俩之间谁更重要是显而易见的。程序员需要我来完成他们的任务。面对现实吧，你还差得远呢。

**接口：**

是吗？我可不这么认为。

**抽象类：**

你就不该妄想你比我更重要。你甚至没有使用真正的继承，只能实现而已。

**接口：**

又来了。接口不使用真正的继承。接口只能实现。真是愚昧。实现哪一点不如继承，实际上实现做得更棒。

**抽象类：**

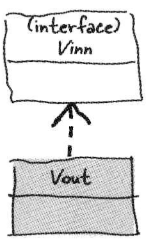
更棒？ 胡说八道。我比你灵活得多。我可以有抽象方法，也可以有具体方法。如果我愿意还可以有虚方法。当然，我不能实例化，但是你也不能呀！还有，常规的类能做的，我都能做到。

**接口：**

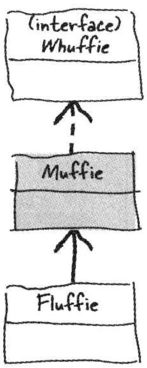
是吗？要让一个类不光继承你还要继承其他类，你办得到吗？你不能同时继承两个类，必须选择到底要继承其中的哪一个类。这太武断了!但是，类能实现的接口个数没有任何限制。你自己说说谁更灵活！程序员用我同样能完成类能做的任何事情。

## Sharpen your pencil Solution

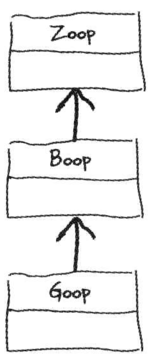
2)



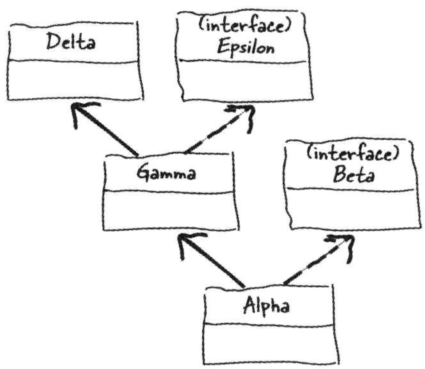

3)




4)




5)




**类图？**

**抽象类：**

你有些夸大你的能力吧。

**接口：**

你以为仅仅因为你能包含代码就是最棒的。但你不能改变这样一个事实，程序员一次只能继承一个类。所以，你是有限制的。当然，我不能包含任何代码，但实际上，对代码有些评价过高。

**抽象类：**

接口就爱说这种傻话。代码当然非常重要！就是因为有代码程序才能运行。

**接口：**

百分之九十的情况下，程序员只希望确保一个对象有某些属性和方法，而不关心它们是怎么实现的。

**抽象类：**

真的吗?我表示怀疑，程序员当然关心属性和方法里有些什么。

**接口：**

就算是吧。但是，程序员编写一个方法时，如果这个方法要使用一个对象，只需要保证这个对象确实有某个方法，至于这个方法具体如何实现并不重要，想想看是不是经常见到这种情况？只要有这个方法就行了，这就够了！程序员只需要写一个接口，问题就迎刃而解！

**抽象类：**

那好，你去告诉一个程序员他不能编写代码。

**接口：**

愿意效劳！

---

2)
```
abstract class Top { }
class Tip : Top { }
```

3)
```
abstract class Fee { }
abstract class Fi : Fee { }
```

4)
```
interface Foo { }
class Bar : Foo { }
class Baz : Bar { }
```

5)
```
interface Zeta { }
class Alpha : Zeta { }
interface Beta { }
class Delta : Alpha, Beta { }
```

Delta继承了Alpha并实现了Beta。

声明？

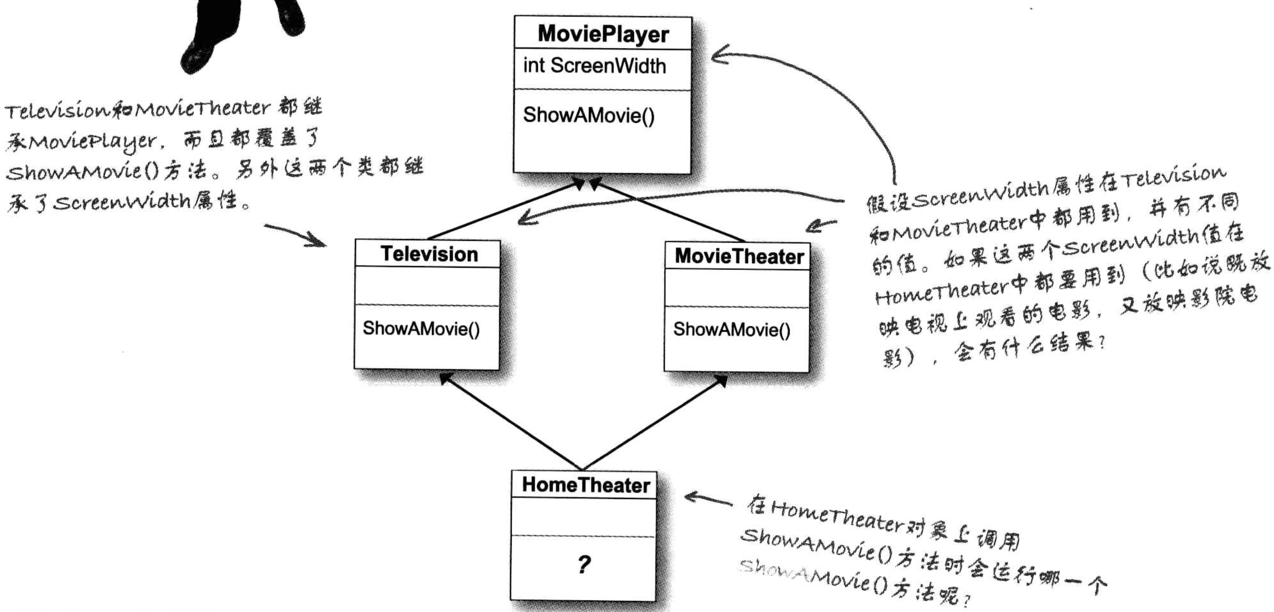


我还是为一个问题很困扰，居然不能继承两个类。如果我不能继承多个类，就必须使用接口。这是C#的一大限制，对不对？

**这不是限制，这是一种保护。**

如果C#允许你继承多个基类，就会为破坏大开方便之门。一种语言允许一个子类继承两个基类时，这称为多重继承（multiple inheritance）。通过提供接口，C#可以保护你不会遇到一个大麻烦，我们喜欢把它称为……

## 致命的死亡菱形




## 避免二义性！

允许这种致命死亡菱形的语言会导致一些很棘手的情况，因为需要一些特殊的规则来处理这种二义性问题……这说明构建程序时需要多做很多工作！C#则通过提供接口来加以保护，使你不必处理这种情况。如果`Television`和`MovieTheater`是接口而不是类，那么一个`ShowAMovie()`方法就能满足二者的需要。接口关心的只是存在一个名为`ShowAMovie()`的方法。



# 池塘谜题

你的任务是从池塘里取出代码片段，在代码和输出中填空。一个代码片段可以使用多次，另外不是所有代码片段都会用到。你的目标是建立一组能编译和运行的类，并得到以下所列的输出。

```
.............. Nose {
    .............. ;
    string Face { get; }
}

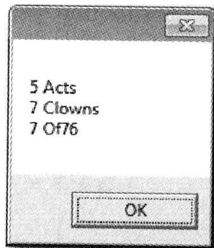
abstract class .......... : .......... {
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
        .............. = face;
    }
    public virtual string Face {
        ....... { .............. ; }
    }
    string face;
}

class .......... : .......... {
    public Clowns() : base("Clowns") { }
}
```

```
class .......... : .......... {
    public Acts() : base("Acts") { }
    public override ............ {
        return 5;
    }
}

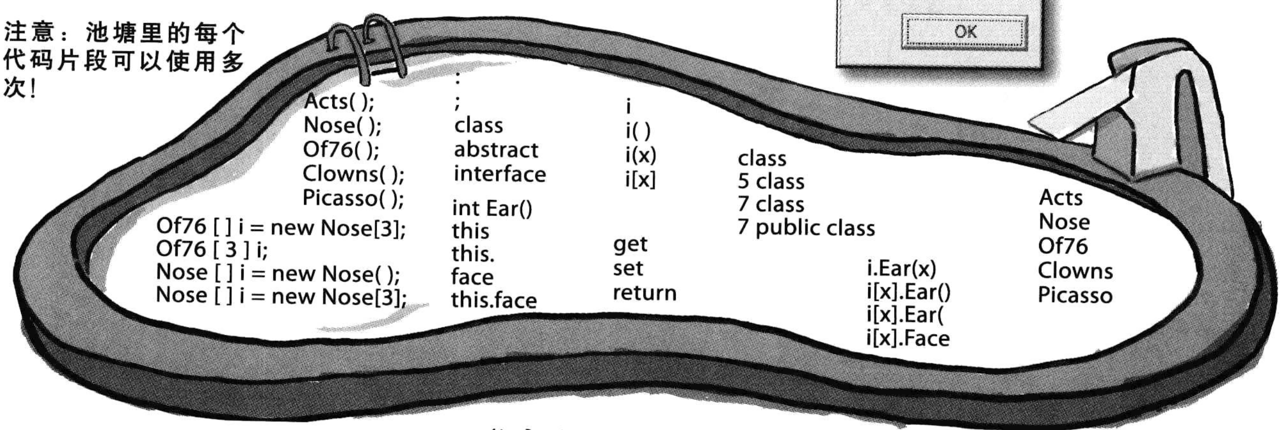
class .......... : .......... {
    public override string Face {
        get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "";
        Nose[] i = new Nose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( .............. + " "
                + .......... ) + "\n";
        }
        MessageBox.Show(result);
    }
}
```

这是入口点，这是一个完整的C#程序。

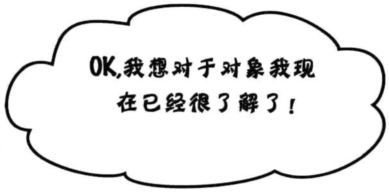


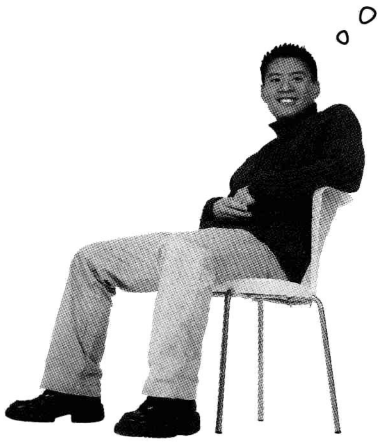

输出

注意：池塘里的每个代码片段可以使用多次！




答案在324页上。



可以把数据和代码结合到类和对象中，首次提出这种想法时绝对是革命性的，不过目前为止，实际上你一直都在用这种方式构建C#程序，所以可以把它认为是普通的编程。

**你是一个面向对象程序员。**

对于你目前所做的工作有一种叫法，称为面向对象程序设计（Object Oriented Programming，OOP）。在C#之类的语言出现之前，人们编写代码时并不使用对象和方法。他们只使用函数（在非OOP程序中称为方法），这些函数都放在一个地方，就好像每个程序就是一个很大的静态类，而且只有静态的方法。解决问题时，采用非OOP方法创建程序对这些问题建模会很困难。幸运的是，你不必采用非OOP方式编写程序，因为OOP正是C#的一个核心部分。

## 面向对象程序设计的4大原则

程序员谈到OOP时，往往是指4个重要原则。你现在应该对这些原则很熟悉了，因为前面一直在用这些原则。前3个原则从名字就很好理解：继承（inheritance）、抽象（abstraction）和封装（encapsulation）。最后一个原则称为多态（polymorphism）。听上去有些奇怪，不过，你会发现你对这个内容同样也已经有所了解。

**继承**

这表示一个类或接口继承另一个类或接口。

**封装**

封装是指创建一个对象，它使用私有字段在内部记录状态，另外使用公共属性和方法，使得其他类只能使用它们需要看到的那部分内部数据。

**抽象**

创建一个类模型时就在使用抽象，首先是更一般（或抽象）的类，然后是从一般类继承的更特定的类。

**多态**

“多态”这个词字面含义是“多种形态”。你能想出在什么情况下代码中一个对象有多种形态？

# 多态表示一个对象可以有多种形态

如果使用一个嘲鸟来取代更一般的动物，或者如果食谱中只需要奶酪，而你用了上好的佛蒙物干酪，这就是在使用多态(polymorphism)。完成向上强制转换和向下强制转换时就采用了多态原则。这表示将一个对象用于需要其他对象的方法或语句中。

**将一个类的实例用在需要其他类型（如父类或该类实现的接口）的语句或方法中，这就是在使用多态。**

## 请特别注意下一个练习中的多态!

接下来要完成一个大练习，这是目前为止最复杂的一个练习，其中将用到大量的多态。所以要特别注意。以下列出了使用多态的4种典型方式。每种方式都提供了一个例子（不过，这些代码行不会出现在练习中）。如果你完成练习时看到类似的代码，请核对这个列表。

☐ 将使用一个类的引用变量设置为等于另一个类的一个实例

```
NectarStinger bertha = new NectarStinger();
INectarCollector gatherer = bertha;
```

☐ 向上强制转换，在一个需要基类的语句或方法中使用一个子类。

```
spot = new Dog();
zooKeeper.FeedAnAnimal(spot);
```

如果FeedAnAnimal()需要一个Animal对象，而Dog继承自Animal，就可以把Dog传入FeedAnAnimal()。

☐ 创建一个引用变量，其类型是一个接口，并把它指向实现该接口的一个对象。

```
IStingPatrol defender = new StingPatrol();
```

这也是向上强制转换!

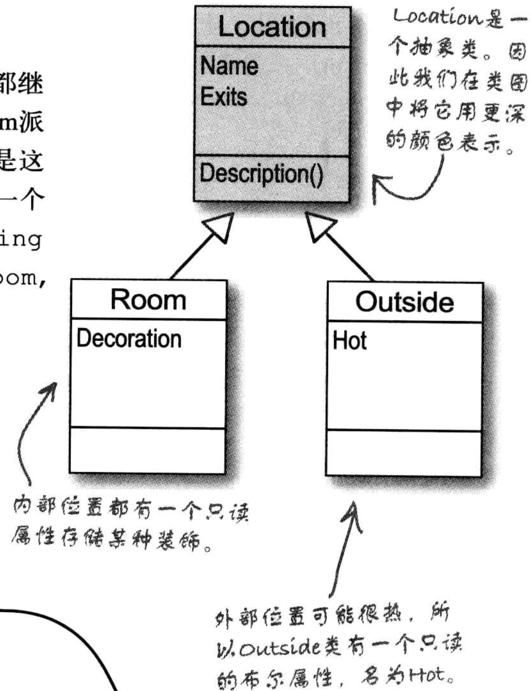
☐ 使用as关键字向下强制转换。

```
void MaintainTheHive(IWorker worker) {
    if (worker is HiveMaintainer) {
        HiveMaintainer maintainer = worker as HiveMaintainer;
        ...
```

MaintainTheHive()方法取任意IWorker作为参数。它使用as将一个HiveMaintainer引用指向worker。

## Long Exercise

下面来建一个房子!创建一个房屋模型，使用类表示房间和位置，并使用一个接口表示所有有门的地方。



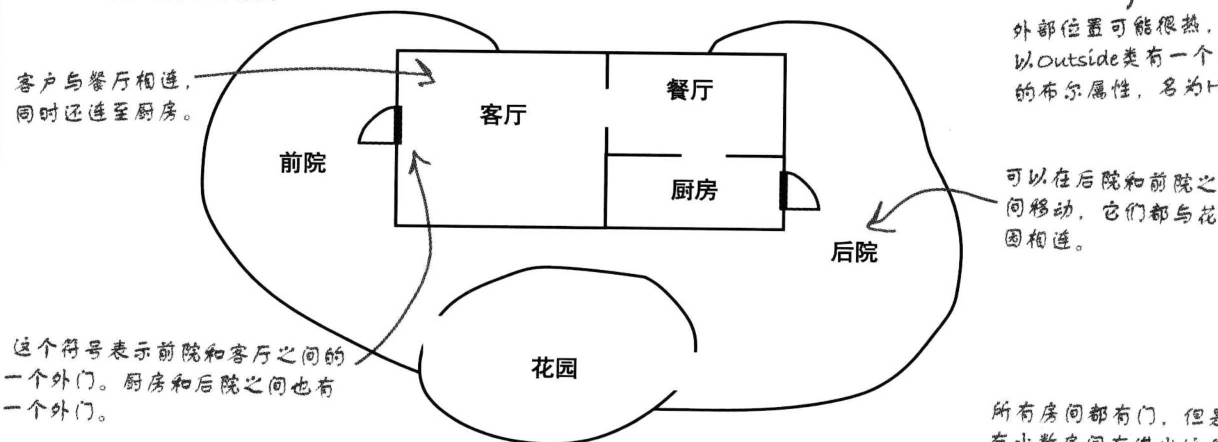

**1** **先从这个类模型开始。**

房子中每个房间或位置都由相应的对象表示。里面的房间都继承自`Room`，外面的位置都从`Outside`继承，Outside与Room派生了同一个基类Location。`Location`有两个字段：Name是这个位置的名字(“Kitchen”)，`Exits`是当前位置连接到的一个`Location`对象数组。所以`diningRoom.Name`将等于“`Dining Room`”，而`diningRoom.Exits`将等于数组`{ LivingRoom, Kitchen }`。

➔**创建一个Windows Application工程，并增加。Location、Room和Outside类。**

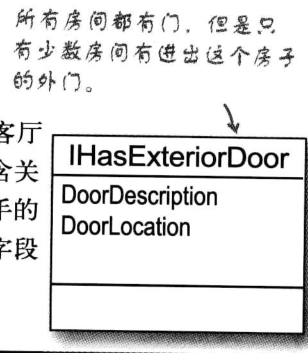
**2** **需要这个房屋的设计蓝图。**

这个房屋有3个房间、一个前院、一个后院，还有一个花园。整个房子有两个门，前门连接客厅与前院，后门连接厨房和后院。




**3** **使用IHasExteriorDoor接口表示有外门的房间。**

这个房子里有两个外门，前门和后门。每个有外门的位置（前院、后院、客厅和厨房）都应当实现`IHasExteriorDoor`。`DoorDescription`只读属性包含关于这个门的描述［前门的描述是“an oak door with a brass knob”（有铜把手的橡木门），后门的描述是“a screen door”（屏风门）］。`DoorLocation`字段包含这个门所通向位置（Location）的一个引用（kitchen）。



**4** **以下是Location类。**

首先给出Location类：

```
abstract class Location {
    public Location(string name) {
        this.name = name;
    }
    public Location[] Exits;
    private string name;
    public string Name {
        get { return name; }
    }
    public virtual string Description {
        get {
            string description = "You're standing in the " + name
                + ". You see exits to the following places: ";
            for (int i = 0; i < Exits.Length; i++) {
                description += " " + Exits[i].Name;
                if (i != Exits.Length - 1)
                    description += ",";
            }
            description += ".";
            return description;
        }
    }
}
```

构造函数设置name字段，这是只读属性Name的后备字段。

Description是一个虚方法。需要覆盖这个方法。

公共字段Exits是一个Location引用数组，其中记录与这个位置相连的所有其他位置。

Room类要覆盖和扩展Description来增加装饰，Outside要增加温度。

Description属性返回描述这个房间的一个串，包括房间名以及与它相连的所有位置的一个列表（存放在Exits[]字段中）。子类需要对这个描述稍做修改，所以要覆盖这个属性。

记住，Location是一个抽象类，可以继承这个类，或者声明Location类型的引用变量，但是不能实例化。

**5** **创建类。**

首先根据类模型创建Room和Outside类。然后再创建两个类：OutsideWithDoor和RoomWithDoor，OutsideWithDoor继承自Outside，并实现IHasExteriorDoor，RoomWithDoor派生Room，并实现IHasExteriorDoor。

以下给出这些类的声明，希望对你有帮助：

现在开始创建这些类，下一页会提供更多有关细节。

```
class OutsideWithDoor : Outside, IHasExteriorDoor
{
    // The DoorLocation property goes here
    // The read-only DoorDescription property goes here
}

class RoomWithDoor : Room, IHasExteriorDoor
{
    // The DoorLocation property goes here
    // The read-only DoorDescription property goes here
}
```

**这是一个相当复杂的练习……不过我们保证这肯定很有意思！等你完成了这个练习后，就会很好地掌握有关的内容。**

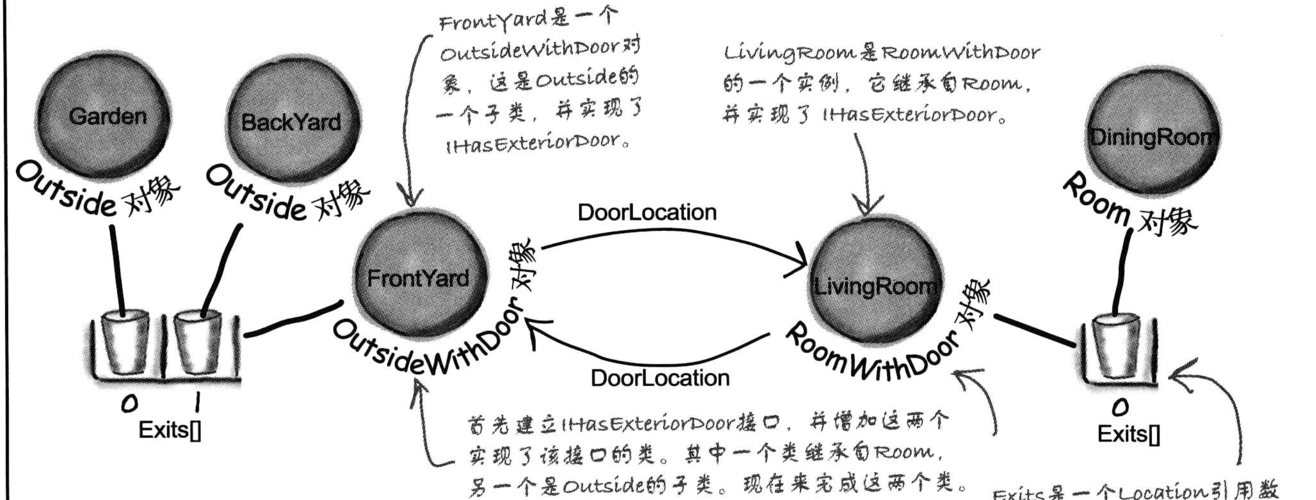
未完，请翻到下一页！

## Long Exercise（续）

既然已经有了类模型，下面可以为房屋的各个部分创建对象，并增加一个窗体来浏览这个房屋。

**6 房屋的对象如何工作。**

以下是两个对象frontYard和diningRoom的体系结构。由于它们都有一个门，所以都必须是实现了IHasExteriorDoor的某个类的实例。DoorLocation属性保存了门另一侧位置的一个引用。




**7 完成这些类的构建，并实例化它们的实例。**

有了这些类之后，下面来具体实现，并建立对象。

★ 需要确保Outside类的构造函数设置只读属性Hot，并覆盖Description属性，如果Hot为true则增加文本“It’s very hot here”。后院很热，但前院和花园不热。

★ Room的构造函数需要设置Decoration，而且要覆盖Description属性，增加“You see (装饰) here”。客厅有一个风格独特的地毯，餐厅有一个水晶装饰灯，厨房有一些不锈钢用具，还有一个通向后院的屏风门。

★ 窗体要创建上述各个对象，并维护各个对象的一个引用。所以向窗体增加一个方法CreateObjects()，并在窗体的构造函数中调用这个方法。

窗体类中对应每个位置都有一个相应的字段。

★ 为房子中的6个位置创建对象，以下是创建其中一个对象的代码：

```
RoomWithDoor livingRoom = new RoomWithDoor( “Living Room”,
    “an antique carpet” , “an oak door with a brass knob”);
```

★ CreateObjects()方法需要填充各对象的Exits[]字段：

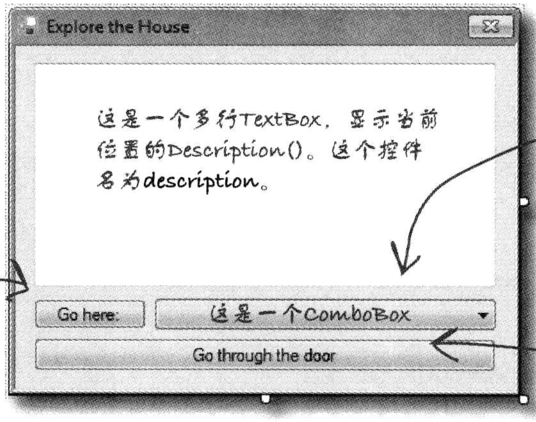
```
frontYard.Exits = new Location[] { backYard, garden };
```

Exits是一个Location引用数组，所以这一行会创建一个数组，其中包含两个引用。

这些是大括号。其他符号都会导致错误。

**8** **建立一个窗体来浏览这个房子。**

建立一个简单的窗体来浏览这个房子。它有一个很大的多行文本框，名为description，用于显示当前房间的描述。还有一个名为exits的ComboBox，会列出当前房间的所有出口。窗体中有两个按钮：点击goHere会转移到ComboBox中选择的那个房间，另外只有当这个房间有一个外门时goThroughTheDoor按钮才可见。




**9** 现在只需让窗体真正发挥作用！

万事俱备，只欠东风，只需把它们集成在一起。

- ★ 窗体中需要有一个名为currentLocation的字段来跟踪当前位置。
- ★ 增加一个MoveToANewLocation()方法，它取一个Location为参数。这个方法首先设置currentLocation为这个新位置。然后使用Items.Clear()方法清空组合框，再使用组合框的Items.Add()方法增加Exits[ ]数组中各个位置的名字。最后，重置组合框，设置其SelectedIndex属性为0来显示列表中的第一项。
- ★ 设置文本框，显示当前位置的描述。
- ★ 使用**is**关键字检查当前位置是否有一个门。如果有，使用“Go through the door”按钮的Visible属性置其可见。如果没有门，则使这个按钮不可见。
- ★ 如果点击“Go here:”按钮，移到组合框中选择的位置。
- ★ 如果点击“Go through the door”按钮，移到与这个门连接的位置。

提示：选择组合框中的一项时，组合框中所选的索引将与Exits[]数组中相应位置的索引相同。

另一个提示：窗体的currentLocation字段是一个Location引用，所以即使它指向一个实现了IHasExteriorDoor的对象，也不能编写类似“currentLocation.DoorLocation”的代码。因为DoorLocation不是Location中的字段。如果确实希望从这个对象得到门所连接的位置，需要完成向下强制转换。

## Long Exercise Solution

以下是对这个房屋建模的有关代码。我们使用类来表示房间和位置，使用接口表示任何有门的地方。

```
interface IHasExteriorDoor {
    string DoorDescription { get; }
    Location DoorLocation { get; set; }
}

class Room : Location {
    private string decoration;

    public Room(string name, string decoration)
        : base(name) {
        this.decoration = decoration;
    }

    public override string Description {
        get {
            return base.Description + " You see " + decoration + ".";
        }
    }
}

class RoomWithDoor : Room, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration, string doorDescription)
        : base(name, decoration)
    {
        this.doorDescription = doorDescription;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }
}
```

这是IHasExteriorDoor接口。

Room类继承了Location，并为只读属性Decoration增加一个后备字段。将由构造函数将设置这个字段。

RoomWithDoor类继承自Room，并实现了IHasExteriorDoor。Room有的特性它都有，另外还为构造函数增加了外门的一个描述。它还增加了DoorLocation，这是门通向的位置的一个引用。DoorDescription和DoorLocation是IHasExteriorDoor要求的属性。

```
class Outside : Location {
    private bool hot;
    public bool Hot { get { return hot; } }

    public Outside(string name, bool hot)
        : base(name)
    {
        this.hot = hot;
    }

    public override string Description {
        get {
            string NewDescription = base.Description;
            if (hot)
                NewDescription += " It's very hot.";
            return NewDescription;
        }
    }
}

class OutsideWithDoor : Outside, IHasExteriorDoor {
    public OutsideWithDoor(string name, bool hot, string doorDescription)
        : base(name, hot)
    {
        this.doorDescription = doorDescription;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }

    public override string Description {
        get {
            return base.Description + " You see " + doorDescription + ".";
        }
    }
}
```

Outside与Room很相似。它继承自Location，并为Hot属性增加了一个后备字段，这个属性将在由基类扩展的Description()方法中用到。

OutsideWithDoor继承自Outside，并实现了IHasExteriorDoor，它与RoomWithDoor非常相似。

基类的Description属性记录这个位置是否热。它根据原Location类的Description属性来增加主描述和相应出口。

未完，翻开下一页！

# Long Exercise Solution（续）

以下是窗体代码。它们都放在Form1.cs中（Form1声明中）。

```
public partial class Form1 : Form
{
    Location currentLocation;

    RoomWithDoor livingRoom;
    Room diningRoom;
    RoomWithDoor kitchen;

    OutsideWithDoor frontYard;
    OutsideWithDoor backYard;
    Outside garden;

    public Form1() {
        InitializeComponent();
        CreateObjects();
        MoveToANewLocation(livingRoom);
    }

    private void CreateObjects() {
        livingRoom = new RoomWithDoor("Living Room", "an antique carpet",
                "an oak door with a brass knob");
        diningRoom = new Room("Dining Room", "a crystal chandelier");
        kitchen = new RoomWithDoor("Kitchen", "stainless steel appliances", "a screen door");

        frontYard = new OutsideWithDoor("Front Yard", false, "an oak door with a brass knob");
        backYard = new OutsideWithDoor("Back Yard", true, "a screen door");
        garden = new Outside("Garden", false);

        diningRoom.Exits = new Location[] { livingRoom, kitchen };
        livingRoom.Exits = new Location[] { diningRoom };
        kitchen.Exits = new Location[] { diningRoom };
        frontYard.Exits = new Location[] { backYard, garden };
        backYard.Exits = new Location[] { frontYard, garden };
        garden.Exits = new Location[] { backYard, frontYard };

        livingRoom.DoorLocation = frontYard;
        frontYard.DoorLocation = livingRoom;

        kitchen.DoorLocation = backYard;
        backYard.DoorLocation = kitchen;
    }
```

窗体用它跟踪在显示哪个房间。

窗体使用这些引用变量跟踪房屋中的各个房间。

窗体的构造函数创建对象，然后使用MoveToANewLocation方法。

窗体创建对象时，首先需要实例化类，并向各个类的构造函数传入适当的信息。

在这里将门的描述传入OutsideWithDoor构造函数。

在这里填充各个实例的Exits[]数组。需要等所有实例都已经创建后才能填充这个数组，否则根本没有具体的内容能够放入各个数组！

需要为IHasExteriorDoor对象设置门通向的位置。

```
private void MoveToANewLocation(Location newLocation) {
    currentLocation = newLocation;

    exits.Items.Clear();
    for (int i = 0; i < currentLocation.Exits.Length; i++)
        exits.Items.Add(currentLocation.Exits[i].Name);
    exits.SelectedIndex = 0;

    description.Text = currentLocation.Description;

    if (currentLocation is IHasExteriorDoor)
        goThroughTheDoor.Visible = true;
    else
        goThroughTheDoor.Visible = false;
}
 private void goHere_Click(object sender, EventArgs e) {
    MoveToANewLocation(currentLocation.Exits[exits.SelectedIndex]);
 }

 private void goThroughTheDoor_Click(object sender, EventArgs e) {
    IHasExteriorDoor hasDoor = currentLocation as IHasExteriorDoor;
    MoveToANewLocation(hasDoor.DoorLocation);
 }
}
```

MoveToANewLocation()方法在窗体中显示一个新位置。

首先需要清空组合框，然后在其中增加各个位置名。最后设置选中的索引（也就是突出显示的一项）为0，这样就会显示列表中的第一项。不要忘记将ComboBox的DropDownStyle属性设置为“DropDownList”，这样一来，用户就不能在组合框中直接键入任何内容，而必须从列表中选择。

如果当前位置没有实现IHasExteriorDoor，这会使“Go through the door”按钮不可见。

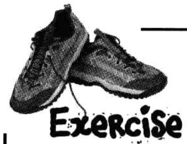
用户点击“Go here:”按钮时，会移到组合框中选中的位置。

需要使用as关键字将currentLocation向下强制转换为一个IHasExteriorDoor，以便访问DoorLocation字段。

## 不过，还没有真正完成！

现在已经能创建一个房屋模型了，不过如果能把它变成一个游戏不是很酷吗？下面就来做这个工作！可以和计算机玩藏猫猫游戏。需要增加一个Opponent类，让他藏在某个房间里。另外把房子做得更大一些。对了，他要有一个藏身的地方！我们将增加一个新接口，这样每个房间都可以有一个藏身的地方。最后，更新窗体，以便检查这些可以藏身的地方，并记录找到对手需要几步。听起来挺有意思吧？当然了！

**开始行动吧！**




来玩藏猫猫!在原来的房屋程序基础上，增加更多房间以及藏身的地方，并增加一个藏起来让你找的对手。

创建一个新工程，并使用IDE的“增加已有项”(Add Existing Item）加入这个练习第一部分的类。

**1 增加一个IHidingPlace接口。**

这里不需要做特别的处理。所有实现了IHidingPlace的Location子类都有一个让对手藏身的地方。只需要一个串来存放藏身地的名字[（“in the closet”（壁橱里），“under the bed”（床底下）等）]。

- ★ 提供一个获取存取方法，不过没有设置存取方法。我们将在构造函数中设置这个属性，因为一旦房间有一个藏身地，就不需要再修改。

**2 增加实现IHidingPlace的类。**

还需要另外两个类：OutsideWithHidingPlace（继承Outside）和RoomWithHidingPlace（继承Room）。另外，让所有有门的房间都有一个藏身地，因此它必须继承RoomWithHidingPlace而不是Room。

有外门的所有房间也要有一个藏身地。

**3 增加一个表示对手的类。**

Opponent对象将在这幢房子里找一个藏身地，你的任务就是找到他。

- ★ 他需要一个私有Location字段（myLocation）来记录他在哪里，还要有一个私有Random字段（random），这在转移到一个随机的藏身地时使用。
- ★ 构造函数取开始位置，将myLocation设置为这个开始位置，并设置random为Random的一个新实例。他先从前院（由窗体传入）开始藏，然后随机地从一个藏身地转移到另一个藏身地。游戏开始后他会转移10次。如果遇到一个外门，他会掷硬币来决定是否通过这个外门。
- ★ 增加一个Move()方法，将对手从其当前位置转移到一个新位置。首先，如果他在一个有门的房间，则掷一个硬币来决定是否通过这道门，所以如果random.Next(2)等于1，则通过这道门。然后随机地选择当前位置的一个出口走过去。如果这里没有藏身地，他会再来一次，从当前位置再选择一个随机的出口走过去，他会反复这样做，直到找到一个地方藏身。
- ★ 增加一个Check()方法，取一个位置作为参数，如果他就藏在那个位置，则返回true，否则返回false。

**4 为这幢房子增加更多房间。**

更新CreateObjects()方法，增加更多房间：

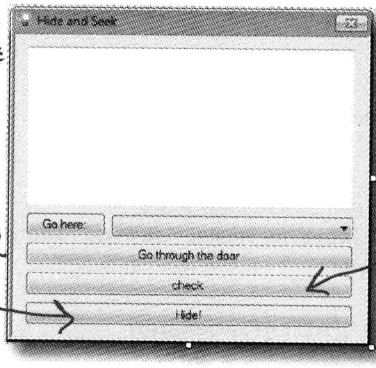
- ★ 增加有栏杆的楼梯，将客厅与楼上的走廊相连，这里有一幅狗的画和一个壁橱可以藏身。
- ★ 楼上的走廊与3个房间相连：有一个大床的主卧室，有一张小床的客卧，还有一个浴室，里面有一个浴缸和一个马桶。两个卧室的床下以及浴缸里都可以藏人。
- ★ 前院和后院都与车道相连，可以藏在车库里。另外，还可以藏在花园的小棚子里。

**5** **好了，可以更新窗体了。**

需要向窗体增加一些按钮。另外，要根据游戏的状态置这些按钮可见或不可见，这会有些麻烦。

上面两个按钮和组合框的用法与前面一样，只不过只是当游戏运行时它们才可见。

中间的按钮名为check。不需要设置它的Text属性。




游戏开始时，hide按钮是唯一可见的按钮。点击这个按钮时，窗体会在文本框中数到10，调用10次对手的Move()方法。再置这个按钮不可见。

这是用来检查这个房间藏身地的按钮。如果所在房间有一个地方可以藏身，这个按钮才可见。如果显示了这个按钮，Text属性会从“check”变为“Check”后面是藏身地的名字，所以如果房间的藏身地在床下，按钮将显示为“Check under the bed”。

**6** **让按钮起作用。**

要为窗体增加两个新按钮。

- ★ 中间的按钮使用对手的`Check()`检查当前房间里的藏身地，只有当所在房间有一个地方可以藏身时这个按钮才可见。如果发现对手，则重置游戏。
- ★ 要使用下面的按钮启动游戏。它会在文本框中从1数到10，首先显示“1……”，等待200毫秒，然后显示“2……”，再显示“3……”，依次数到10。每数一个数后，调用对手的Move()方法让对手转移。然后显示半秒钟“Ready or not, here I come!”（准备好了吗？我来了！”），游戏开始。

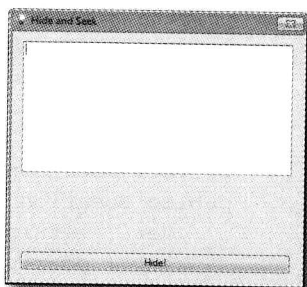
翻回第2章，复习一下DoEvents()和Sleep()，这里将用到这两个方法。

**7** **增加一个方法重绘窗体，另外增加一个方法重置游戏。**

增加一个`RedrawForm()`方法在description文本框中放入正确的文本，适当地设置按钮为可见或不可见，并在中间按钮上设置正确的标签。然后增加一个`ResetGame()`方法，找到对手时会运行这个方法。它会重置对手对象，让他再从前院开始藏，点击“Hide!”按钮时他会开始藏。此时，窗体上除了文本框和“Hide!”按钮可见外，其余都不可见。文本框应当指出在哪里找到对手，以及对手转移了多少次。

**8** **跟踪玩家转移了多少次。**

文本框要显示玩家检查藏身地或在房间之间转移了多少次。找到对手时，他会弹出一个消息框，指出“You found me in X moves!”（你找到我用了X步！）。




**9** **启动程序时保证一切正常。**

第一次启动程序时，只能看到一个空的文本框和“Hide!”按钮。点击这个按钮时，游戏开始了！

**Exercise Solution**

在原来的房屋程序基础上，增加更多房间以及藏身的地方，并增加一个藏起来让你找的对手。

这是新的IHidingPlace接口。它只有一个string字段，有一个获取存取方法返回藏身地的名字。

```
interface IHidingPlace {
    string HidingPlaceName { get; }
}

class RoomWithHidingPlace : Room, IHidingPlace {
    public RoomWithHidingPlace(string name, string decoration, string hidingPlaceName)
        : base(name, decoration)
    {
        this.hidingPlaceName = hidingPlaceName;
    }

    private string hidingPlaceName;
    public string HidingPlaceName {
        get { return hidingPlaceName; }
    }

    public override string Description {
        get {
            return base.Description + " Someone could hide " + hidingPlaceName + ".";
        }
    }
}
```

RoomWithHidingPlace类继承自Room，实现了IHidingPlace，还增加了HidingPlaceName属性。构造函数设置了它的后备字段。

```
class RoomWithDoor : RoomWithHidingPlace, IHasExteriorDoor {
    public RoomWithDoor(string name, string decoration,
                        string hidingPlaceName, string doorDescription)
        : base(name, decoration, hidingPlaceName)
    {
        this.doorDescription = doorDescription;
    }

    private string doorDescription;
    public string DoorDescription {
        get { return doorDescription; }
    }

    private Location doorLocation;
    public Location DoorLocation {
        get { return doorLocation; }
        set { doorLocation = value; }
    }
}
```

因为我们认为每个有门的房间还需要一个藏身地，所以让RoomWithDoor继承RoomWithHidingPlace。对它只需要做一个修改，就是构造函数再增加一个藏身地名参数，并把它进一步发送给RoomWithHidingPlace构造函数。

```
class OutsideWithHidingPlace : Outside, IHidingPlace {
    public OutsideWithHidingPlace(string name, bool hot, string hidingPlaceName)
        : base(name, hot)
    { this.hidingPlaceName = hidingPlaceName; }

    private string hidingPlaceName;
    public string HidingPlaceName {
        get { return hidingPlaceName; }
    }

    public override string Description {
        get {
            return base.Description + " Someone could hide " + hidingPlaceName + ".";
        }
    }
}

class Opponent {
    private Random random;
    private Location myLocation;
    public Opponent(Location startingLocation) {
        myLocation = startingLocation;
        random = new Random();
    }
    public void Move() {
        if (myLocation is IHasExteriorDoor) {
            IHasExteriorDoor LocationWithDoor =
                                myLocation as IHasExteriorDoor;
            if (random.Next(2) == 1)
                myLocation = LocationWithDoor.DoorLocation;
        }
        bool hidden = false;
        while (!hidden) {
            int rand = random.Next(myLocation.Exits.Length);
            myLocation = myLocation.Exits[rand];
            if (myLocation is IHidingPlace)
                hidden = true;
        }
    }
    public bool Check(Location locationToCheck) {
        if (locationToCheck != myLocation)
            return false;
        else
            return true;
    }
}
```

OutsideWithHidingPlace类继承自Outside，并且与RoomWithHidingPlace一样，实现了IHidingPlace。

Opponent类构造函数取一个开始位置作为参数。它创建Random的一个新实例，用来在房间之间随机转移。

Move()方法首先使用is关键字检查当前房间是否有一个门，如果有，则有50%的可能性通过这道门。然后转移到一个随机位置，并继续转移，直到找到一个藏身地。

Move()方法的关键是这个while循环。它会一直循环，直到变量hidden为true，如果找到一个有藏身地的房间则置其为true。

Check()方法只是检查对手的位置是否与使用Location引用传入的位置相同。如果它们指向同一个对象，就找到了对手！

未完，翻到下一页！

**Exercise Solution**

这些是窗体的代码。只有goHere_Click()和goThroughTheDoor_Click()方法保持不变。

这些都是Form1类的字段。Form1使用这些字段来记录位置、对手以及玩家走了几步。

```
int Moves;

Location currentLocation;

RoomWithDoor livingRoom;
RoomWithHidingPlace diningRoom;
RoomWithDoor kitchen;
Room stairs;
RoomWithHidingPlace hallway;
RoomWithHidingPlace bathroom;
RoomWithHidingPlace masterBedroom;
RoomWithHidingPlace secondBedroom;

OutsideWithDoor frontYard;
OutsideWithDoor backYard;
OutsideWithHidingPlace garden;
OutsideWithHidingPlace driveway;

Opponent opponent;
```

Form1构造函数创建对象、创建对手，然后重置游戏。我们为ResetGame()增加了一个布尔参数，从而只在获胜时显示消息，而在第一次启动程序时不显示。

```
public Form1() {
    InitializeComponent();
    CreateObjects();
    opponent = new Opponent(frontYard);
    ResetGame(false);
}

private void MoveToANewLocation(Location newLocation) {
    Moves++;
    currentLocation = newLocation;
    RedrawForm();
}

private void RedrawForm() {
    exits.Items.Clear();
    for (int i = 0; i < currentLocation.Exits.Length; i++)
        exits.Items.Add(currentLocation.Exits[i].Name);
    exits.SelectedIndex = 0;
    description.Text = currentLocation.Description + "\r\n(move #" + Moves + ")";
    if (currentLocation is IHidingPlace) {
        IHidingPlace hidingPlace = currentLocation as IHidingPlace;
        check.Text = "Check " + hidingPlace.HidingPlaceName;
        check.Visible = true;
    }
    else
        check.Visible = false;
    if (currentLocation is IHasExteriorDoor)
        goThroughTheDoor.Visible = true;
    else
        goThroughTheDoor.Visible = false;
}
```

MoveToANewLocation()方法设置新位置，然后重新绘制窗体。

我们需要藏身地名，但是只得到了CurrentLocation对象，它并没有HidingPlaceName属性，所以可以使用as把这个引用向下强制转换为一个IHidingPlace变量。

RedrawForm()填充组合框列表，设置文本（增加移动的步数），然后根据是否有门或者房间是否有一个藏身处设置按钮为可见或不可见。

哇，只需增加这么几行代码就可以向房间增加一层楼！正是因为这个原因，所以说封装得很好的类和对象会非常有用。

```
private void CreateObjects() {
    livingRoom = new RoomWithDoor("Living Room", "an antique carpet",
            "inside the closet", "an oak door with a brass handle");
    diningRoom = new RoomWithHidingPlace("Dining Room", "a crystal chandelier",
             "in the tall armoire");
    kitchen = new RoomWithDoor("Kitchen", "stainless steel appliances",
            "in the cabinet", "a screen door");
    stairs = new Room("Stairs", "a wooden bannister");
    hallway = new RoomWithHidingPlace("Upstairs Hallway", "a picture of a dog",
            "in the closet");
    bathroom = new RoomWithHidingPlace("Bathroom", "a sink and a toilet",
            "in the shower");
    masterBedroom = new RoomWithHidingPlace("Master Bedroom", "a large bed",
            "under the bed");
    secondBedroom = new RoomWithHidingPlace("Second Bedroom", "a small bed",
            "under the bed");

    frontYard = new OutsideWithDoor("Front Yard", false, "a heavy-looking oak door");
    backYard = new OutsideWithDoor("Back Yard", true, "a screen door");
    garden = new OutsideWithHidingPlace("Garden", false, "inside the shed");
    driveway = new OutsideWithHidingPlace("Driveway", true, "in the garage");

    diningRoom.Exits = new Location[] { livingRoom, kitchen };
    livingRoom.Exits = new Location[] { diningRoom, stairs };
    kitchen.Exits = new Location[] { diningRoom };
    stairs.Exits = new Location[] { livingRoom, hallway };
    hallway.Exits = new Location[] { stairs, bathroom, masterBedroom, secondBedroom };
    bathroom.Exits = new Location[] { hallway };
    masterBedroom.Exits = new Location[] { hallway };
    secondBedroom.Exits = new Location[] { hallway };
    frontYard.Exits = new Location[] { backYard, garden, driveway };
    backYard.Exits = new Location[] { frontYard, garden, driveway };
    garden.Exits = new Location[] { backYard, frontYard };
    driveway.Exits = new Location[] { backYard, frontYard };

    livingRoom.DoorLocation = frontYard;
    frontYard.DoorLocation = livingRoom;

    kitchen.DoorLocation = backYard;
    backYard.DoorLocation = kitchen;
}
```

这个新CreateObjects()方法创建了这座房子所需的全部有对象。这很像原来的老方法，不过现在的房间比原来多得多。

未完，翻到下一页！

## Exercise Solution（续）

这是窗体其余的代码。goHere和goThroughTheDoor按钮事件处理程序与这个练习第一部分中原来的按钮事件处理程序是一样的，可以翻回去几页来查看这两个方法。

```
private void ResetGame(bool displayMessage) {
    if (displayMessage) {
        MessageBox.Show("You found me in " + Moves + " moves!");
        IHidingPlace foundLocation = currentLocation as IHidingPlace;
        description.Text = "You found your opponent in " + Moves
              + " moves! He was hiding " + foundLocation.HidingPlaceName + ".";
    }
    Moves = 0;
    hide.Visible = true;
    goHere.Visible = false;
    check.Visible = false;
    goThroughTheDoor.Visible = false;
    exits.Visible = false;
}

private void check_Click(object sender, EventArgs e) {
    Moves++;
    if (opponent.Check(currentLocation))
        ResetGame(true);
    else
        RedrawForm();
}

private void hide_Click(object sender, EventArgs e) {
    hide.Visible = false;

    for (int i = 1; i <= 10; i++) {
        opponent.Move();
        description.Text = i + " ... ";
        Application.DoEvents();
        System.Threading.Thread.Sleep(200);
    }

    description.Text = "Ready or not, here I come!";
    Application.DoEvents();
    System.Threading.Thread.Sleep(500);

    goHere.Visible = true;
    exits.Visible = true;
    MoveToANewLocation(livingRoom);
}
```

ResetGame()方法重置游戏。它显示最后的消息，然后将“Hide!”以外的其余按钮都置为不可见。

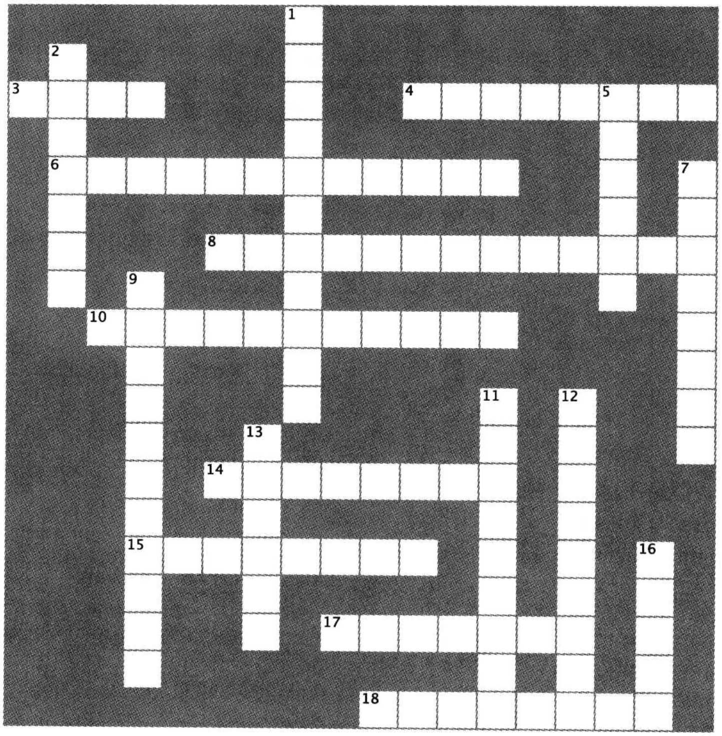
我们希望显示藏身处的名字，但是CurrentLocation是一个Location引用，所以不能通过它访问HidingPlaceName字段。幸运的是，可以使用as关键字把它向下强制转换为一个指向同一个对象的IHidingPlace引用。

点击check按钮时，它会检查对手是否藏在当前房间里。如果确实藏在这里，则重置游戏。如果没有，则重新绘制窗体（更新移动的步数）。

还记得第2章FlashyThing中的DoEvents()吗？如果没有这个方法，文本框不会自行刷新，程序看起来就像被冻住了。

通过点击hide按钮启动游戏。首先它会让自己（hide按钮）不可见，然后从1数到10，让对手转移。最后使第一个按钮和组合框可见，再让玩家从客厅开始找。MoveToANewLocation()方法调用了RedrawForm()。

# OOP填字游戏




## 横向

3.抽象方法所没有的。

4. C#不允许__________继承。

6.将一个子类传递到需要其基类的方法，就是在使用这个OOP原则。

8. 这个OOP原则要求隐藏私有数据，只公开其他类需要访问的方法和字段。

10.这是4大OOP原则之一，需要使用冒号操作符实现。

14. 接口中的每个方法都自动为__________ 。

15. 如果一个类实现了一个接口，而这个接口又__________了另一个接口，那么类也要实现它的所有成员。

17. 接口内不允许放置这个访问修饰符。

18.面向对象（Object__________ ）编程是指创建程序将数据和代码组合到类和对象中。

## 纵向

1.将公共方法从特定类移到它们都继承的一个一般类，就是在使用这个OOP原则。

2. 如果实现了一个接口的类没有具体实现接口的所有方法、获取存取方法和设置存取方法，则工程不能__________。

5.接口中的所有方法都自动作为__________方法。

7. 抽象类可以包含抽象和__________方法。

9.不能__________抽象类。

11. 实现了__________的类必须包含它定义的所有方法、获取存取方法和设置存取方法。

12. 能够__________接口。

13. 如果一个__________实现了一个接口，is关键字返回true。

16.理论上接口不包含__________，但是可以定义获取存取方法和设置存取方法，所以从外部看就像有一样。



# 305页池塘谜题答案

你的任务是从池塘里取出代码片段，在代码和输出中填空。一个代码片段可以使用多次，另外不是所有代码片段都会用到。你的目标是建立一组能编译和运行的类，并得到以下所列的输出。

```
interface Nose {
    int Ear() ;
    string Face { get; }
}

abstract class Picasso : Nose {
    public virtual int Ear()
    {
        return 7;
    }
    public Picasso(string face)
    {
        this.face = face;
    }
    public virtual string Face {
        get { return face ; }
    }
    string face;
}

class Clowns : Picasso {
    public Clowns() : base("Clowns") { }
}
```

属性可以出现在类中的任何位置。如果放在代码的最上面，代码将更可读，不过将face属性放在Picasso类的最下面也是完全合法的。

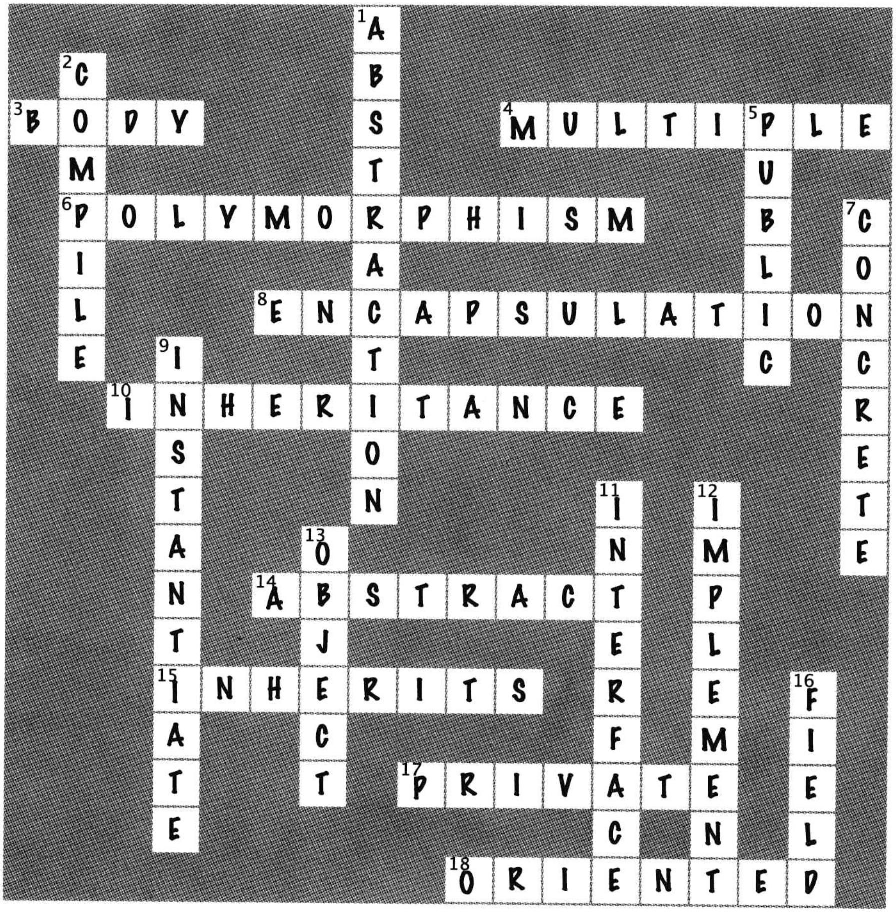
Acts类在这里调用Picasso的构造函数（Acts类继承自Picasso）。它把“Acts”传入构造函数，这会存储在face属性中。

```
class Acts : Picasso {
    public Acts() : base("Acts") { }
    public override int Ear() {
        return 5;
    }
}

class Of76 : Clowns {
    public override string Face {
        get { return "Of76"; }
    }
    public static void Main(string[] args) {
        string result = "" ;
        Nose[] i = new Nose[3];
        i[0] = new Acts();
        i[1] = new Clowns();
        i[2] = new Of76();
        for (int x = 0; x < 3; x++) {
            result += ( i[x].Ear() + " "
                + i[x].Face ) + "\n";
        }
        MessageBox.Show(result);
    }
}
```

Face是一个获取存取方法，返回face属性的值。它们都在Picasso中定义，并由子类继承。

# OOP填字游戏答案


1 ABSTRACTION
2 COMPILE
3 BODY
4 MULTIPLE
5 PUBLIC
6 POLYMORPHISM
7 CONCRETE
8 ENCAPSULATION
9 INSTANTIATE
10 INHERITANCE
11 INTERFACE
12 IMPLEMENT
13 OBJECT
14 ABSTRACT
15 INHERITS
16 FIELD
17 PRIVATE
18 ORIENTED

# 8 枚举与集合

# 存储大量数据






**简直是倾盆大雨。**

在真实世界里，我们处理的数据绝对不会是一点点。实际上，数据会成包、成捆，甚至成堆地压过来。我们需要一些非常强大的工具来整理所有这些数据，这就引入了集合。利用集合，可以存储程序需要处理的所有数据，并进行排序和管理。这样一来，你可以专心考虑编写程序处理数据，而由集合负责为你记录数据。

# 字符串并不总适用于存储所有类型的数据

假设有一些工蜂，都由Worker类表示。要编写一个以任务作为参数的构造函数该怎么做呢？如果使用字符串作为任务名，最后可能会得到如下的代码：

我们的蜜蜂管理软件会用一个字符串（如“Sting Patrol”或“Nectar Collector”等等）记录每个工蜂的工作。

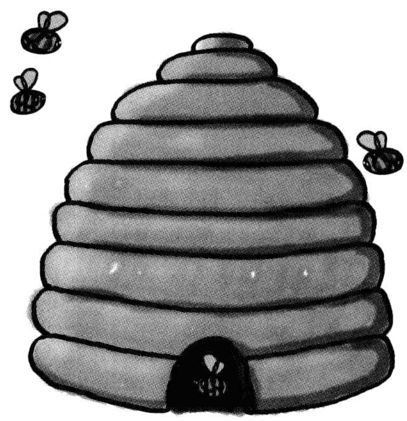
尽管程序只支持Sting Patrol（带刺巡逻）、Nectar Collector（花露采集）以及蜜蜂能做的另外一些工作，但这个代码确实允许向构造函数传入下面这些值。

```
Worker buzz = new Worker("Attorney General");
Worker clover = new Worker("Dog Walker");
Worker gladys = new Worker("Newscaster");
```

这个代码能顺利编译，没有任何问题。不过，这些工作对于蜜蜂来说毫无意义。实际上Worker类就不应该将这些值作为合法数据。

针对这个问题，你可能会在Worker构造函数中增加代码，检查各个字符串，确保它是一个合法的、蜜蜂确实能完成的工作。不过，如果要增加蜜蜂能做的新工作，就必须修改这个代码，并且重新编译Worker类。但是这是一个目光短浅的方案。倘若其他类需要检查工蜂能做哪些工作呢？这样一来，就需要编写重复的代码，这条路可不怎么样。

我们需要这样一种办法，指出“嘿，这里只允许几个特定的值”。我们需要枚举出允许使用的值。



# Enum枚举允许处理一组合法值

enum数据类型只允许某个数据取某些特定的值。所以，可以定义一个名为Jobs的枚举，并定义所允许的一些工作：

大括号里的内容称为枚举项列表（enumerator list），其中每一项称为枚举项（enumerator）。整体则称为枚举（enumeration）。

但是大多数人只称之为enum。

这是enum的名。

```
enum Job {
    NectarCollector,
    StingPatrol,
    HiveMaintenance,
    BabyBeeTutoring,
    EggCare,
    HoneyManufacturing,
}
```

每一项都是一个合法的工作。它们都可以用作一个Jobs值。

最后一个枚举项后面不必有逗号，不过如果在这里加一个逗号，使用剪切粘贴重新排列这些枚举项时会更容易。

将各个值用一个逗号分隔，最后对整个枚举加一个大括号。

现在，可以像这样引用这些值：

这是enum的名。

这就是你希望由枚举得到的值。

```
Worker nanny = new Worker(Job.EggCare);
```

我们已经修改了Worker构造函数，使它接受Worker.Jobs作为它的参数类型。

但是不能为enum直接造一个新值。如果硬要如此，程序将不能编译。

```
private void button1_Click(object sender EventArgs e)
{
        Worker buzz = new Worker(Jobs.AttorneyGeneral);
}
```

将从编译器得到这个错误。

```
'Jobs' does not contain a definition for
'AttorneyGeneral'
```

# 通过枚举，可以用名来表示数字

有时，如果对应数字指定了相应的名，处理起来会更容易。可以为enum中的值指定数字，并使用名来引用这些值。这样一来，代码中就不会充斥着大量无法解释的数字了。下面这个enum记录了犬类大赛中各种技巧的得分。

**可以把一个int强制转换为一个enum，还可以把一个（基于int的）enum强调转换回int。**

```
public enum TrickScore {
    Sit = 7,
    Beg = 25,
    RollOver = 50,
    Fetch = 10,
    ComeHere = 5,
    Speak = 30,
}
```

这些项并没有特定的顺序，另外对同一个数字可以指定多个名。

提供一个名，然后是"="，再后面是这个名所表示的数字。

有些enum使用另外一种不同的类型，如byte或long，如本页最下面的enum，这些enum也可以转换回原来的类型。

(int)强制类型转换告诉编译器把这个名转换为它表示的数字。所以由于TrickScroe.Fetch的索引为10，(int)TrickScore.Fetch就会转换为int值10。

这是从使用TrickScore enum的方法中摘取的一个片段，这里将它强制转换为一个int，然后又强制转换回来。

```
int value = (int)TrickScore.Fetch * 3;
MessageBox.Show(value.ToString());
TrickScore score = (TrickScore)value;
MessageBox.Show(score.ToString());
```

由于Fetch值为10，这条语句会把值设置为30。

可以把int强制转换回一个TrickScore。因为value等于30，score会设置为TrickScore.Fetch。所以调用score.ToString()时，它会返回"Fetch"。

可以把enum强制转换为一个数，并用它完成某些计算，或者可以使用`ToString()`方法把名处理为一个字符串。如果没有为名指定任何数字，默认地会为列表中的各项指定一个值。第一项指定值为0，第二项指定1，以此类推。

但是，如果希望某个枚举项使用非常大的数怎么办呢？enum中数字的默认类型为int，所以需要使用：操作符指定所需的类型，如下所示：

```
public enum TrickScore : long {
        Sit = 7,
        Beg = 2500000000025
}
```

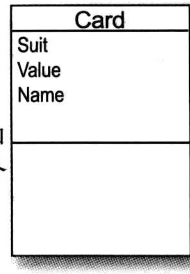
这告诉编译器要把TrickScores enum中的值处理为long而不是int。

如果试图编译这个代码，但没有指定类型为long，就会得到以下消息：

`Cannot implicitly convert type 'long' to 'int'.`




使用前面关于枚举学到的知识构建一个存储扑克牌的类。




1. **创建一个新工程，并增加一个Card类。**
需要两个公共字段：`Suit`（可以是`Spades`、`Clubs`、`Diamonds`或`Hearts`）和`Value`（`Ace, Two, Three ... Ten, Jack, Queen, King`）。另外需要一个只读属性`Name`（“`Ace of Spades`”，“`Five of Diamonds`”）。

2. **使用两个enum来定义花色和牌面大小。**
在IDE中使用我们熟悉的“Add”→“Class”特性来增加这些字段。确保`(int)Card.Suits.Spades`等于0，后面是`Clubs`（等于1），`Diamonds(2)`和`Hearts(3)`。让Value等于牌面大小：`(int)Card.Values.Ace`应当等于1，`Two`应当是2，`Three`是3等。`Jack`应当等于11，`Queen`等于12，`King`是13。

3. **为牌名增加一个属性。**
Name应当是一个只读属性。获取存取方法应当返回一个描述这张牌的串。这个代码在窗体中运行，要调用Card类的Name属性，并显示：

```
Card card = new Card(Suits.Spades, Values.Ace);
string cardName = card.Name;
```

为了正常完成工作，Card类需要一个有两个参数的构造函数。

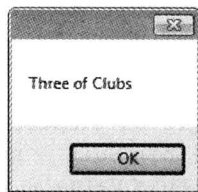
`cardName`的值应当是“`Ace of Spades`”。

4. **增加一个窗体按钮，弹出消息框随机显示一张牌的牌名。**
可以让程序随机创建一张牌，其花色和大小随机确定，将一个介于0～3的随机数强制转换为一个`Suits`，并将另一个介于1到13的随机数强制转换为`Values`。为此，可以利用内置`Random`类的一个特性，使用它提供的3种不同方式来调用`Next()`方法：

如果调用一个方法有多种方式，这称为重载。稍后会更多地讨论这个内容……

```
Random random = new Random();
int numberBetween0and3 = random.Next(4);
int numberBetween1and13 = random.Next(1, 14);
int anyRandomInteger = random.Next();
```

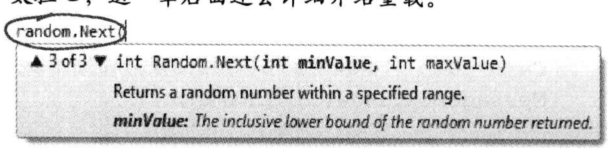
这就告诉Random返回一个最小为1但不超过14的值。




## there are no Dumb Questions

**问：** 等一下。键入这个代码时，我注意到，使用这个`Random.Next()`方法时弹出了一个智能提示窗口，说什么“3 of 3”，这是什么意思？

**答：** 你看到的是一个重载方法。如果类有一个方法，而你可以采用多种方式调用这个方法，这就称为重载。使用一个有重载方法的类时，IDE会让你知道所有可能的选择。在这里，`Random`类有3个可能的`Next()`方法。一旦在代码窗口中键入“random.Next(”，IDE就会弹出它的智能提示框，显示出各个不同重载方法的参数。“3 of 3”旁边的上下箭头允许你在这3个重载方法间上下滚动。如果处理一个有数十个重载定义的方法，这会非常有用。所以写代码时，一定要确保选择了正确的重载`Next()`方法！但是现在不用太担心，这一章后面还会详细介绍重载。



**Exercise Solution**

对值加以限制很重要，在这方面，一副牌就是一个很好的例子。没有人希望把牌翻过来后看到Joker of Club或13 of Hearts。如下编写Card类。

```
enum Suits {
    Spades,
    Clubs,
    Diamonds,
    Hearts
}
```

没有指定值时，列表中的第1项就等于0，第2项等于1，第3项等于2，以此类推。

```
enum Values {
    Ace = 1,
    Two = 2,
    Three = 3,
    Four = 4,
    Five = 5,
    Six = 6,
    Seven = 7,
    Eight = 8,
    Nine = 9,
    Ten = 10,
    Jack = 11,
    Queen = 12,
    King = 13
}
```

在这里将Values.Ace的值设置为1。

Card类有一个类型为Suits的Suit属性，另外有一个类型为Values的Value属性。

```
class Card {
    public Suits Suit { get; set; }
    public Values Value { get; set; }

    public Card(Suits suit, Values value) {
        this.Suit = suit;
        this.Value = value;
    }

    public string Name {
        get { return Value.ToString() + " of " + Suit.ToString(); }
    }
}
```

Name属性的获取存取方法可以利用这一点，即enum的ToString()方法会返回enum的名并将其转换为一个串。

这是按钮的代码，它会显示一张随机牌的牌名。

在这里使用重载的Random.Next()方法生成一个随机数，我们将把它强制转换为enum。

```
Random random = new Random();
private void button1_Click(object sender, EventArgs e) {
    Card card = new Card((Suits)random.Next(4), (Values)random.Next(1, 14));
    MessageBox.Show(card.Name);
}
```

## 可以使用数组来创建一副牌……

如果想创建一个类来表示一副牌，该怎么做？需要一种办法来记录这副牌中的每一张牌，而且需要知道这些牌的顺序是怎样的。Card数组就能做到这一点，这副牌中最上面的牌索引为0，下一张牌的索引为1，以此类推。以下代码可以作为起点，最初Deck是包含52张牌的整副牌。

这个数组声明会继续，直到声明完这副牌中的所有牌。这里省略只是为了节省篇幅。

```
class Deck {
    private Card[] cards =  {
        new Card(Suits.Spades, Values.Ace),
        new Card(Suits.Spades, Values.Two),
        new Card(Suits.Spades, Values.Three),
        // ...
        new Card(Suits.Diamonds, Values.Queen),
        new Card(Suits.Diamonds, Values.King),
    };

    public void PrintCards() {
        for (int i = 0; i < cards.Length; i++)
            Console.WriteLine(cards[i].Name());
    }
}
```

## ……但是，如果你还想做其他呢？

不过，想想看还可能怎样处理这副牌。打牌时，通常需要换牌的顺序，增加牌或出牌。要使用一个数组来做到这一切可不那么容易。

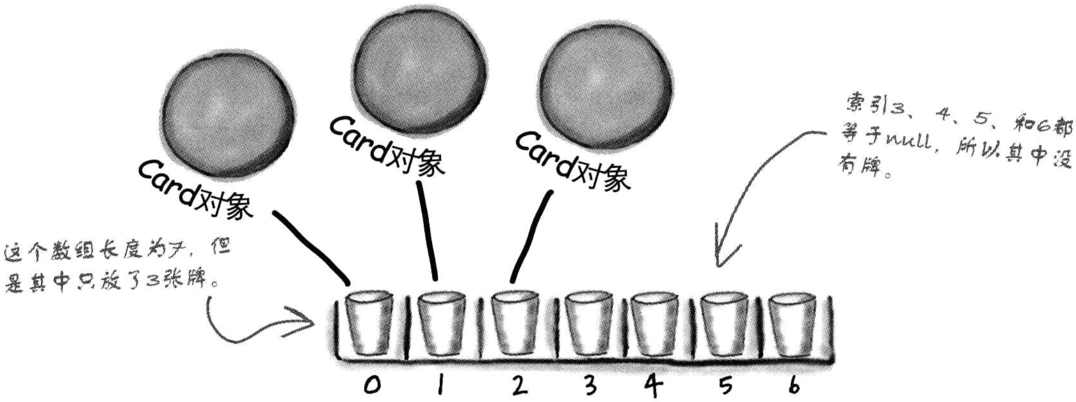
**BRAIN POWER**

怎样为Deck类增加一个Shuffle()方法，以随机的顺序重排这副牌（也就是洗牌）？要增加一个方法出这副牌最上面的第一张牌，又该怎么做？如果要加一张牌呢？

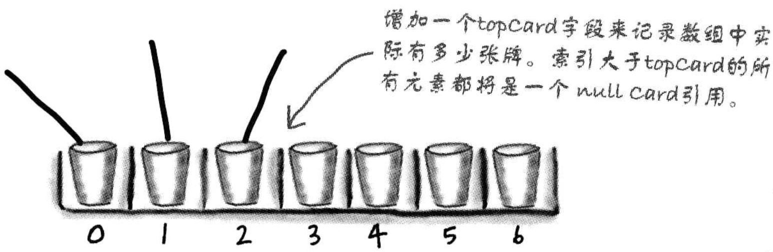
# 数组很难处理

用数组来存储一个固定的值或引用列表还不错，但是如果需要移动数组元素，或者要增加更多元素而数组无法容纳，问题就有些麻烦了。

1. 每个数组都有一个长度，需要知道这个长度才能正常地处理数组。可以使用null引用将某些数组元素置为空：




2. 需要跟踪放了多少张牌。因此需要一个int字段，可以称之为topCard，其中存放数组中最后一张牌的索引。所以对于以上包含3张牌的数组，它的长度为7，但topCard要设置为2。




实际上.NET Framework中内置有一个Array.Resize()方法可以完成同样的工作。

3. 不过，现在问题开始复杂了。增加一个Peek()方法返回最上面一张牌的引用，以便看一眼这张牌，这很容易。但是如果希望增加一张牌呢？如果topCard小于数组的Length-1，只需将这张牌增加到数组中该索引的位置，并把topCard增1。不过，如果这个数组已经满了，就需要创建一个新的、更大的数组，并把现在的牌复制到这个新数组中。删除一张牌很容易，不过，将topCard减1后，需要确保对应已删除那张牌的数组索引应当将元素再设置为null。如果需要从这个列表的中间删除一张牌呢？要删除第4张牌，就需要把第5张牌移回来替换原来的第4张牌，再移动第6张取代原来第5张的位置，再移动第7张牌……天哪，真是麻烦！

# 利用List可以很容易地存储……任何事物的集合

.NET Framework提供了大量集合（collection）类，可以处理增加和删除数组元素时遇到的所有这些麻烦。最常用的一种集合就是List<T>。一旦创建了List<T>对象，将很容易增加项，从列表中任何位置删除项，查看某一项，甚至可以很容易地把一项从列表中的一个位置移到另一个位置。列表的工作如下：

有时我们去省略<T>，因为加上<T>会让这本书有些不便阅读。看到List时，要把它想成是List<T>！

**1** **首先创建List<T>的新实例。**

每个数组都有一个类型，不能只是说有一个数组，而应该说一个int数组或是Card数组等等。List也是一样。需要指定列表中能存放的对象或值的类型，为此，使用new关键字创建List时，要把类型放在一对尖括号<>中。

```
List<Card> cards = new List<Card>();
```




创建列表时指定了<Card>，所以现在这个列表只能存放Card对象的引用。




**List<T>末尾的<T>表示这是泛型。**

T会替换为一个类型，所以List<int>就表示int的List。后面几页会有大量关于泛型的练习。

**2** **现在可以为List<T>增加元素。**

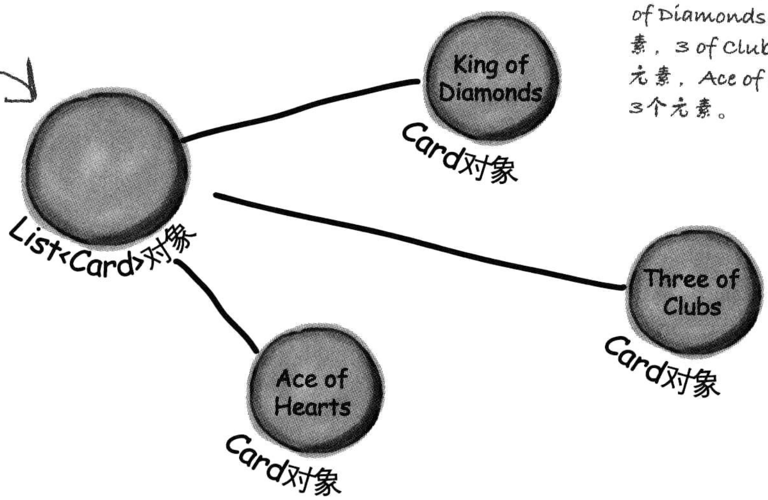
一旦得到一个List<T>对象，想要增加多少项就可以向这个列表增加多少项（只要项的类型与创建新List<T>时指定的类型匹配）。

这表示可以对它们赋值的类型：接口、抽象类、基类等。

```
cards.Add(new Card(Suits.Diamonds, Values.King);
cards.Add(new Card(Suits.Clubs, Values.Three);
cards.Add(new Card(Suits.Hearts, Values.Ace);
```

列表会维护元素的顺序，这与数组类似。King of Diamonds是第1个元素，3 of Clubs是第2个元素，Ace of Hearts是第3个元素。

可以根据需要向List增加很多元素，只需调用它的Add()方法。List会确保为这些项取得了足够的“楼”（空间）。如果空间用完了，它会自动调整大小。



# List比数组更灵活

List类是.NET Framework内置的类，利用List可以对对象做很多处理，而其中很多工作使用原来普通的数组是做不到的。看看利用`List<T>`能做哪些事情。

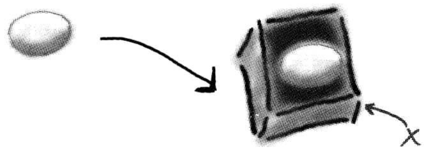
**1 可以建立一个列表。**

```
List<Egg> myCarton = new List<Egg>();
```




**2 在其中增加元素。**

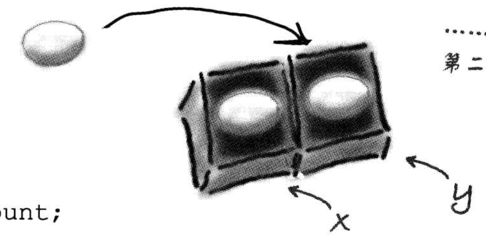
```
Egg x = new Egg();
myCarton.Add(x);
```




**3 增加其他元素。**

```
Egg y = new Egg();

myCarton.Add(y);
```




**4 查看其中有哪些元素。**

```
int theSize = myCarton.Count;
```

现在在列表中搜索Egg。当然这会返回true。

**5 查看其中是否有某个特定的元素。**

```
bool Isin = myCarton.Contains(x);
```

x的索引将是0，y的索引将是1。

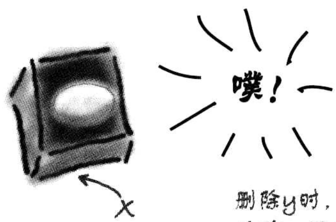
**6 确定它在哪个位置上。**

```
int idx = myCarton.IndexOf(y);
```

这会告诉你列表自行调整大小之前它能容纳的对象个数。

**7 从中取出某个元素。**

```
myCarton.Remove(y);
```




删除y时，List中只剩下x，所以列表将收缩！而且最后会垃圾回收。

## Sharpen your pencil

填完下表，查看左边的List代码，如果使用常规的数组你认为应该写什么代码，把相应的代码填在右边。我们并不指望你能全部做对，尽量好好猜一猜。

假设这些语句都按顺序执行。

我们已经为你填了几处……

| List | 常规数组 |
|---|---|
| List<String> myList = new List <String>(); | String [] myList = new String[2]; |
| | |
| String a = "Yay!"; | String a = "Yay!"; |
| myList.Add(a); | |
| | |
| String b = "Bummer"; | String b = "Bummer"; |
| myList.Add(b); | |
| | |
| int theSize = myList.Count; | |
| | |
| Guy o = myList[1]; | |
| | |
| bool isIn = myList.Contains(b); <br><br> 提示：这里需要多于一行代码。→ | |

## Sharpen your pencil Solution

你的任务是填完下表，查看左边的List代码，如果使用常规的数组你认为应该写什么代码，把相应的代码填在右边。

| List | 常规数组 |
|---|---|
| `List<String> myList =`<br>`new List <String>();` | `String[] myList = new String[2];` |
| `String a = "Yay!"`<br>`myList.Add(a);` | `String a = "Yay!";`<br>`myList[0] = a;` |
| `String b = "Bummer";`<br>`myList.Add(b);` | `String b = "Bummer";`<br>`myList[1] = b;` |
| `int theSize = myList.Count;` | `int theSize = myList.Length;` |
| `Guy o = myList[1];` | `Guy o = myList[1];` |
| `bool isIn = myList.Contains(b);` | `bool isIn = false;`<br>`for (int i = 0; i < myList.`<br>`Length; i++) {`<br>`if (b == myList[i]) {`<br>`isIn = true;`<br>`}`<br>`}` |

List也是对象，与目前为止使用的所有其他类一样，它也以同样的方式使用方法。在IDE中只需在List名旁边键入一个“.”，就可以看到一组可用的方法。类似于你自己创建的类，可以用同样的方式为这些方法传入参数。

使用数组就受限得多。创建数组时需要设置数组的大小，而且对数组完成的所有逻辑都必须由你自己来编写。

.NET Framework确实提供了一个Array类，能让某些事情稍微容易一些…… 不过我们还是重点强调List对象，因为List使用要容易得多。

# List会动态伸缩

动手做！

List的一大特色是，创建List时你不需要知道它有多长。List会自动伸缩来适应它的内容。以下是一些list方法的例子，正是因为有这样一些方法，使得List的处理比数组要容易得多：创建一个新的控制台应用，并把以下代码增加到Main()。这不会打印任何结果，需要使用调试工具单步跟踪代码来查看做了些什么。

这里声明一个Shoe对象的List，名为ShoeCloset。

```
List<Shoe> shoeCloset = new List<Shoe>();

shoeCloset.Add(new Shoe()
    { Style = Style.Sneakers, Color = "Black" });
shoeCloset.Add(new Shoe()
    { Style = Style.Clogs, Color = "Brown" });
shoeCloset.Add(new Shoe()
    { Style = Style.Wingtips, Color = "Black" });
shoeCloset.Add(new Shoe()
    { Style = Style.Loafers, Color = "White" });
shoeCloset.Add(new Shoe()
    { Style = Style.Loafers, Color = "Red" });
shoeCloset.Add(new Shoe()
    { Style = Style.Sneakers, Color = "Green" });

int numberOfShoes = shoeCloset.Count;
foreach (Shoe shoe in shoeCloset) {
    shoe.Style = Style.Flipflops;
    shoe.Color = "Orange";
}

shoeCloset.RemoveAt(4);

Shoe thirdShoe = shoeCloset[3];
Shoe secondShoe = shoeCloset[2];
shoeCloset.Clear();

shoeCloset.Add(thirdShoe);
if (shoeCloset.Contains(secondShoe))
    Console.WriteLine("That's surprising.");
```

可以在List.Add()方法内部使用new语句。

**foreach是一种针对List的特殊循环。它会对List中的每个对象执行一些语句。这个循环创建一个名为shoe的标识符。循环处理各项时，先设置shoe等于列表中的第一项，然后设置为第二项，再是第三项，直到循环结束。**

foreach循环也能处理数组！实际上，foreach循环适用于任何集合。

这会返回List中Shoe对象的总数。

这个foreach循环依次处理鞋柜里的每个鞋。

Remove()方法根据对象引用删除一个对象，RemoveAt()则根据索引号删除对象。

Clear()方法删除一个List中的所有对象。

清空列表之前我们保存了两个鞋的引用。把其中一个再增加回列表中，另一个会消失。

这一行不会运行，因为Contains()会返回false。我们只向先前清空的列表中增加了thirdShoe，而没有增加fifthShoe。

以下是我们使用的Shoe类，以及这个类使用的Style enum。

```
class Shoe {
    public Style Style;
    public string Color;
}

enum Style {
    Sneakers,
    Loafers,
    Sandals,
    Flipflops,
    Wingtips,
    Clogs,
}
```

# 泛型可以存储任何类型

你已经看到List可以存储串或Shoe。还可以建立整数List，或者为所创建的任何其他对象建立列表。这就使List成为一个泛型集合（generic collection）。创建一个新的List对象时，会把它与一个特定类型绑定：可以有int列表、串列表或者Shoe对象列表。这就使得处理List很容易，一旦创建列表，总能知道列表中数据的类型。

这并不是指增加字母T。这只是一种记法，只要一个类或接口能使用所有类型时就会看到这种记法。<T>是指可以在这里放置一个类型，如List<Shoe>，并限制成员只能是这种类型。

```
List<T> name = new List<T>();
```

List可以非常灵活（允许任何类型），也可以非常受限。所以它们不仅可以完成数组能做的所有工作，而且还能做得更多。

.NET Framework提供了一些泛型接口，允许你建立的集合处理任何类型。List实现了这些接口，正是因为这个原因，就像处理Shoe对象List一样，可以用完全相同的方式创建整数List。

→ 你可以自己试试看。在IDE中键入List，然后右键点击它，并选择“Go To Definition”。就会转到List类的声明。它实现了以下接口：

RemoveAt()，IndexOf()和Insert()就来自这里。

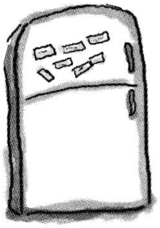
```
class List<T> : IList<T>,
ICollection<T>, IEnumerable<T>, IList,
ICollection, IEnumerable
```

Add()、Clear()、CopyTo()和Remove()都来自这个接口。这是所有泛型集合的基础。

这个接口允许对使用foreach循环，还提供了其他一些功能。

## BULLET POINTS

- **List**是.NET Framework中的一个类。
- List会动态调整为所需的大小。它有一定的容量，一旦向列表增加的数据足够多，列表就会扩展以适应新数据。
- 向List中放入元素可以使用**Add()**。从List删除元素使用**Remove()**。
- 可以使用RemoveAt()根据对象的索引号删除对象。
- 要使用一个类型参数来声明List的类型，这是一个类型名，放在尖括号里。例如，List<Frog>表示List只能存放Frog类型的对象。
- 要查找List中某个对象在什么位置（或者是否存在这个对象），可以使用**IndexOf()**。
- 要得到一个List中的元素个数，可以使用**count**属性。
- 可以使用**Contains()**方法来得出一个List中是否有某个特定对象。
- **foreach**是一种特殊类型的循环，它会迭代处理一个List中的所有元素，并对各元素执行某些代码。**foreach**循环的语法是：**foreach** (string s in **StringList**)。不必明确告诉**foreach**循环增1，它会自己负责循环处理整个List。



# 代码磁条

你能重新组织这些代码片段吗？要求建立一个能正常工作的Windows Form，点击按钮时它能弹出以下消息框。

```
a.RemoveAt(2);
```

```
List<string> a = new List<string>();
```

```
public void printL (List<string> a){
```

```
if (a.Contains("two")) {
    a.Add(twopointtwo);
}
```

```
a.Add(zilch);
a.Add(first);
a.Add(second);
a.Add(third);
```

```
}
```

```
string result = "";
```

```
if (a.Contains("three")){
        a.Add("four");
    }
```

```
foreach (string element in a)
{
    result += "\n" + element;
}
```

```
MessageBox.Show(result);
```

```
}
```

```
if (a.IndexOf("four") != 4) {
        a.Add(fourth);
    }
```

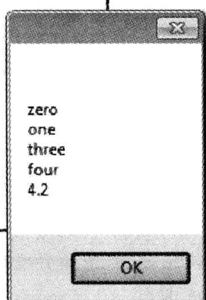
```
printL(a);
```

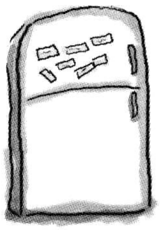
```
string zilch = "zero";
string first =  "one";
string second = "two";
string third = "three";
string fourth = "4.2";
string twopointtwo = "2.2";
```

```
private void button1_Click(object sender,
EventArgs e){



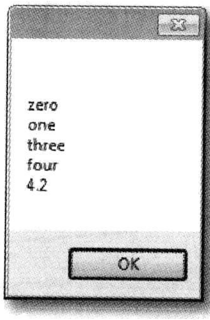
}
```





# 代码磁条答案

**还记得第3章我们讨论过要使用直观的名字吧？没错，这样可以得到好的代码，但是这也会让这些题目太过容易。　不过在实际编程中千万不要用像“printL()”这样晦涩的名字！**




```
private void button1_Click(object sender, EventArgs e)
{
    List<string> a = new List<string>();
    string zilch = "zero";
    string first = "one";
    string second = "two";
    string third = "three";
    string fourth = "4.2";
    string twopointtwo = "2.2";
    a.Add(zilch);
    a.Add(first);
    a.Add(second);
    a.Add(third);
    if (a.Contains("three")){
        a.Add("four");
    }
    a.RemoveAt(2);
    if (a.IndexOf("four") != 4) {
        a.Add(fourth);
    }
    if (a.Contains("two")) {
        a.Add(twopointtwo);
    }
    printL(a);
}
public void printL (List<string> a){
    string result = "";
    foreach (string element in a)
    {
        result += "\n" + element;
    }
    MessageBox.Show(result);
}
}
```

尽管这里声明了“2.2”，但它没有增加到列表中，你能解释为什么吗？

RemoveAt()删除索引为#2的元素，这是列表中的第3个元素。

printL()方法使用一个foreach循环来迭代处理一个字符串列表，将各个字符串增加到一个大字符串中，然后在一个消息框中显示这个结果字符串。

foreach循环迭代处理列表中的所有元素，并显示。

## there are no Dumb Questions

**问：为什么要使用enum而不是List呢？它们不是解决同样的问题吗？**

**答：**Enum与List稍有一点区别。最重要的是，enum是类型，而List是对象。

可以把enum认为是存储常量列表的一种简便方法，以便按名来访问各个元素。enum对于保证代码可读性很有意义，另外可以确保使用正确的变量名来访问经常用到的值。

List可以存储任何对象。由于这是一个对象列表，列表中的每个元素可以有它自己的方法和属性。而enum不同，enum必须指定C#中的某种值类型（如第4章第一页上的类型）。所以，不能在enum中存储引用变量。

另外，enum不能动态改变大小。枚举不能实现接口，也不能有方法，必须将其强制转换为另一种类型才能在其他变量中存储enum的一个值。把以上几点综合起来，你应该了解到这两种存储数据的方法之间存在一些明显的差异。不过它们本身都非常有用。

**问：OK, 听上去List的功能相当强大。那为什么有时还想使用数组呢？**

**答：**如果你知道要处理的项数是固定的，或者你想要一个固定的定长值序列，数组就很完美。幸运的是，使用ToArray() 方法可以很容易地把一个列表转换为数组…… 另外可以使用List<T>的某个重载构造函数把一个数组转换为列表。

数组还会使程序占用更少的内存，需要的CPU时间也更少，但这只会带来很小的一点性能提升。如果要在一秒内把同一件事做几百万次，你可能更愿意使用数组而不是列表。不过，如果你的程序运行很慢，仅仅靠从列表转换为数组不太可能解决这个问题。

**问：我不懂“泛型（generic）”是什么意思。为什么要称它是一个泛型集合呢？为什么数组不是泛型集合？**

**答：**泛型集合（generic collection）是一个集合对象（或一个内置对象，可以存储和管理大量其他对象），这种集合只能存储一种类型（或者可以有多种类型，稍后就会看到）。

**问：嗯，这解释了“集合”部分，那为什么是“泛型”呢？**

**答：**超市经常用大包装装东西，包装标牌上只指出了其中商品的名称（“薯片”、“可乐”、“肥皂”等）。这些通用标牌的重点是包装里的东西，而不是它如何显示。

泛型数据类型（generic data types）也是一样。不论其中是什么类型，List<T>的工作都完全一样。Shoe对象、Card对象、int、long甚至其他List的List都相当于容器级。所以可以增加、删除、插入元素等，而不论列表里到底是什么。

“泛型”（generic）是指，尽管List的一个特定实例只能存储一种特定类型，但List类适用于任何类型。

这正是<T>的含义。通过这种方法可以将List的一个特定实例绑定到某种类型。不过，List类本身完全可以处理任何类型。正是因为这一点，泛型集合与我们之前看到的类型是不同的。

**问：能不能有无类型的列表？**

**答：**不能。每一个列表，实际上，每个泛型集合（稍后还会了解其他泛型集合）都必须有一个与之关联的类型。C#确实有一些非泛型列表，名为ArrayList，可以存储任何类型的对象。如果想使用ArrayList，需要在代码中包含一行“using System.Collections;”。不过你不需要这么做，因为List<object>完全能胜任！

**创建一个新的List对象时，总要提供一个类型，告诉C#它会存储什么类型的数据。列表可以存储值类型（int、bool或string），也可以存储一个类。**

# 集合初始化方法类似于对象初始化方法

C#提供了一个很不错的简便方法，可以减少创建列表并在其中增加大量数据项所需的代码。创建新的List对象时，可以使用一个集合初始化方法（collection initializer）来提供初始的一组数据项。列表一创建就会增加这些数据项。

前几页上应该已经见过这个代码，它创建一个新的List<Shoe>，并填入一些新的Shoe对象。

```
List<Shoe> shoeCloset = new List<Shoe>();
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Black" });
shoeCloset.Add(new Shoe() { Style = Style.Clogs, Color = "Brown" });
shoeCloset.Add(new Shoe() { Style = Style.Wingtips, Color = "Black" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "White" });
shoeCloset.Add(new Shoe() { Style = Style.Loafers, Color = "Red" });
shoeCloset.Add(new Shoe() { Style = Style.Sneakers, Color = "Green" });
```

用一个集合初始化方法重写这个代码

注意到每个Shoe对象如何用自己的对象初始化方法来初始化吗？你可以把它们嵌套在一个集合初始化方法中，就像这样。

创建集合初始化方法时，可以取原先用Add()增加的各项，把它们增加到创建列表的语句中。

```
List<Shoe> shoeCloset = new List<Shoe>() {
    new Shoe() { Style = Style.Sneakers, Color = "Black" },
    new Shoe() { Style = Style.Clogs, Color = "Brown" },
    new Shoe() { Style = Style.Wingtips, Color = "Black" },
    new Shoe() { Style = Style.Loafers, Color = "White" },
    new Shoe() { Style = Style.Loafers, Color = "Red" },
    new Shoe() { Style = Style.Sneakers, Color = "Green" },
};
```

创建列表的语句后面是一对大括号，其中包含各个new语句，各new语句之间用逗号分隔。

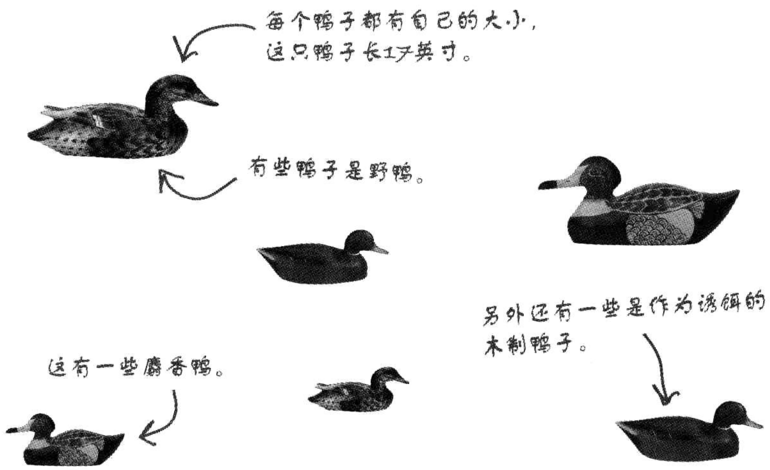
初始化方法中并不仅限于使用new语句，还可以包含变量。

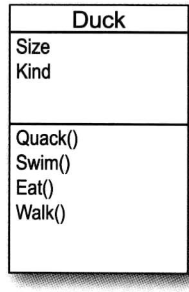
**集合初始化方法可以使代码更紧凑，能够把创建列表和增加一组初始项结合在一起。**

# 下面来创建一个Duck列表

动手做！

这里是一个Duck类，用于记录你收集的鸭子（你确实收集鸭子，是不是）。创建一个新的控制台应用，并增加一个新的Duck类和一个KindOfDuck enum。




```
class Duck {
    public int Size;
    public KindOfDuck Kind;
}

enum KindOfDuck {
    Mallard,
    Muscovy,
    Decoy,
}
```

这个类有两个公共字段。它还有一些方法，在这里没有显示。

我们将使用一个名为KindOfDuck的enum来记录集合中有哪些类型的鸭子。

将Duck和KindOfDuck增加到工程中。

## 以下是Duck列表的初始化方法

我们有6只鸭子，所以创建一个List<Duck>，它的集合初始化方法中有6个语句。初始化方法中的每个语句创建一个新鸭子，使用一个对象初始化方法设置各个Duck对象的Size和Kind字段。把这个代码增加到program.cs的Main()方法中。

```
List<Duck> ducks = new List<Duck>() {
    new Duck() { Kind = KindOfDuck.Mallard, Size = 17 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 18 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 14 },
    new Duck() { Kind = KindOfDuck.Muscovy, Size = 11 },
    new Duck() { Kind = KindOfDuck.Mallard, Size = 14 },
    new Duck() { Kind = KindOfDuck.Decoy, Size = 13 },
};

// 这会防止输出在你读到之前就消失。
Console.ReadKey();
```

要在Main()方法中增加代码来输出到控制台。一定要在最后增加这行代码，这样程序就会一直打开，直至你按下某个按键。

# 列表很容易，但排序可能很困难

要对数字或字母排序可能并不难。不过如果两个对象要排序该怎么做？特别是如果它们有多个字段呢？某些情况下，你可能希望按一个name字段的值对对象排序，但另外一些情况下，可能需要根据高度或出生日期完成对象的排序。完成排序的方法很多，List支持所有这些方法。

### 可以按大小对鸭子列表排序……



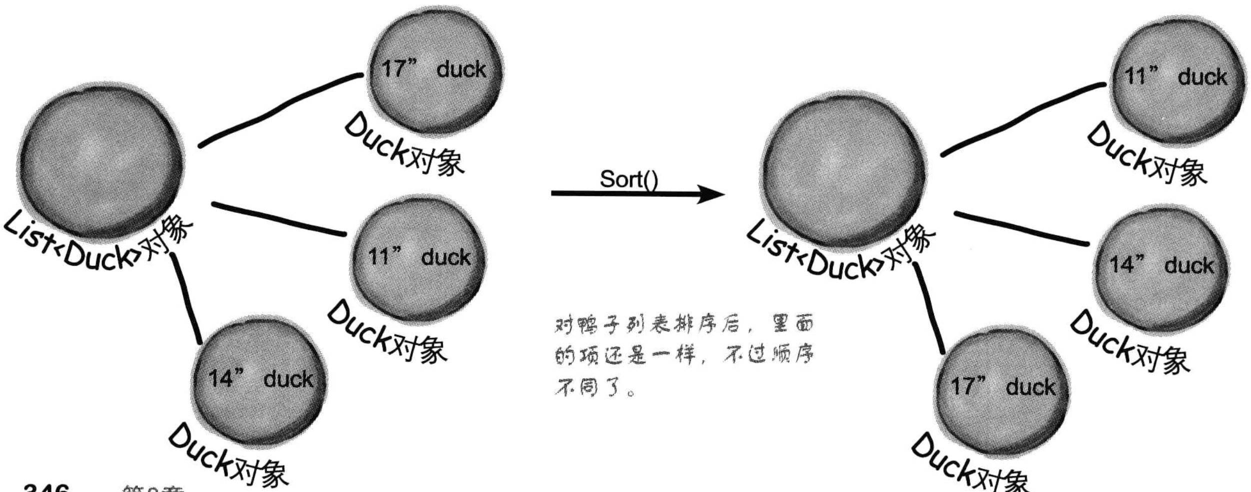

### ……或者按类型




## List知道如何排序

技术上讲，并不是List<T>自己知道如何排序。它要依靠一个IComparer<T>对象，稍后就会介绍有关内容。

每个列表都提供了一个`Sort()`方法，它会重排列表中的所有项，使它们有序排列。List已经知道如何对大多数内置类型和类进行排序，要教它学会如何对你的定制类进行排序也很容易。



# IComparable<Duck>帮助列表完成鸭子排序

List.Sort()方法知道如何对实现了**IComparable<T>**接口的类或类型排序。这个接口只有一个成员，即CompareTo()方法。Sort()使用一个对象的CompareTo()方法与其他对象比较，并使用其返回值（一个int）来确定哪一个在前。

但是有时需要对某个对象列表排序，而这些对象并没有实现IComparable接口，对此.NET有另一个接口来提供帮助。可以向Sort()传入实现了IComparer<T>的一个类的实例。这个接口也只有一个方法。List的Sort()方法使用这个比较对象的Compare()方法来比较一对对象，从而得出在有序列表中哪一个对象在前面。

**要让List的内置Sort()方法能够处理某个类，只需让这个类实现IComparable<T>接口，并增加一个CompareTo()方法。**

## 对象的CompareTo()方法将它自己与另一个对象比较

要对List对象排序，一种办法是修改Duck类，让它实现IComparable<Duck>接口。为此，需要增加一个CompareTo()方法，取一个Duck引用作为参数。如果要比较的鸭子（作为参数传入的鸭子）应当出现在有序表中当前鸭子的后面，CompareTo()会返回一个正数。

下面更新工程的Duck类，让它实现IComparable<Duck>，从而根据鸭子大小完成排序：

```
class Duck : IComparable<Duck> {
    public int Size;
    public KindOfDuck Kind;

    public int CompareTo(Duck duckToCompare) {
        if (this.Size > duckToCompare.Size)
            return 1;
        else if (this.Size < duckToCompare.Size)
            return -1;
        else
            return 0;
    }
}
```

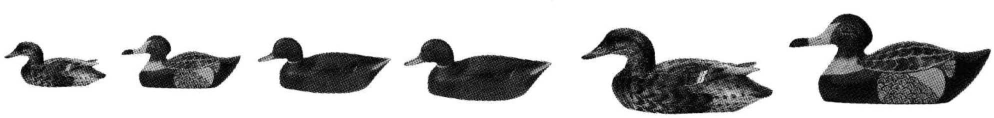
实现IComparable<T>时，要指定让类实现这个接口所要比较的类型。

大多数CompareTo()方法都与此类似。这个方法首先将当前鸭子的Size字段与另一个鸭子的Size字段比较。如果当前鸭子更大一些，则返回1。如果当前鸭子较小，则返回-1。如果两只鸭子同样大，就返回0。

如果想从小到大对列表排序，当前鸭子与一个更小的鸭子比较时，让CompareTo()返回一个负数；如果与一个更大的鸭子比较，则返回一个正数。

把这个代码增加到Main()方法的最后，但放在Console.ReadKey()调用上面，告诉鸭子列表自行排序。可以在 CompareTo()方法中加一个断点，使用调试工具查看它的具体工作。

```
ducks.Sort();
```



# 使用IComparer告诉List如何排序

List有一个特殊的.NET Framework内置接口，允许你构建一个单独的类来帮助List<T>完成成员的排序。通过实现IComparer<T>接口，可以告诉List你希望如何对对象排序。为此，要实现IComparer<T>接口中的Compare()方法。它取两个对象参数x和y，并返回一个int。如果x小于y，这个方法应当返回一个负值。如果二者相等，则返回0。如果x大于y，方法要返回一个正值。

**取决于如何实现IComparer<T>，List会以不同的方式实现排序。**

下面是一个例子，展示了如何声明一个比较类从而按大小比较Duck对象。把它作为一个新类增加到工程中：

这个类实现了IComparer，并指定能够进行排序的对象类型：Duck对象。

```
class DuckComparerBySize : IComparer<Duck>
{
        public int Compare(Duck x, Duck y)
        {
            if (x.Size < y.Size)
                return -1;
            if (x.Size > y.Size)
                return 1;
            return 0;
        }
}
```

这些类型总是一致的：各参数有相同的类型。

Compare()方法返回一个int，并有两个参数（均为待排序类型）。

在这个方法中可以完成你希望的比较。

任何负数表示对象x应当在对象y前面。x“小于”y。

任何正数表示对象x应当在对象y后面。x“大于”y。

0表示（使用这个比较计算时）应当认为这两个对象相等。

这个方法会打印List<Duck>中的鸭子。

把这个PrintDucks方法增加到这个工程的Program类中，从而可以打印列表中的鸭子。

更新Main()方法，在对列表排序之前和之后分别调用这个方法，来看看有什么结果！

```
public static void PrintDucks(List<Duck> ducks)
{
    foreach (Duck duck in ducks)
        Console.WriteLine(duck.Size.ToString() + "-inch " + duck.Kind.ToString());
    Console.WriteLine("End of ducks!");
}
```

# 创建比较对象的一个实例

要使用IComparer<T>排序时，需要创建实现了这个接口的类的一个新实例。这个对象存在的意义就是帮助List.Sort()确定如何对数组排序。不过类似于其他（非静态）类，使用之前需要先实例化。

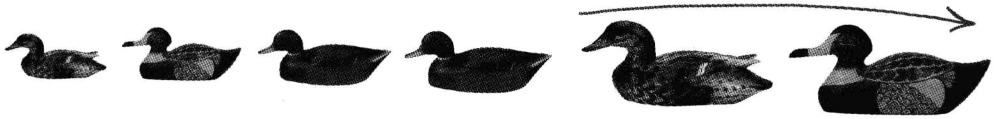
这里没有列出前几页已有的初始化列表的代码。在对列表排序之前一定要先初始化列表！如果没有初始化，你会得到一个null指针异常。

```
DuckComparerBySize sizeComparer = new DuckComparerBySize();
ducks.Sort(sizeComparer);
PrintDucks(ducks);
```

要向Sort()传递这个新的DuckComparerBySize对象实例作为参数。

把这个代码增加到程序的Main()方法，来看这些鸭子如何排序。

从小到大排序……



## 多个IComparer实现，多种对象排序方法

可以创建多个IComparer<Duck>类，提供不同的排序逻辑采用不同方法对鸭子排序。需要以某种特定方式排序时，可以调用相应的比较类。以下是另一个鸭子比较类实现，可以增加到你的工程中：

这个比较类按鸭子种类排序。要记住，比较enum Kind时，就是在比较其索引值。

所以Mallard在Muscovy之前，而Muscovy在Decoy前面。

```
class DuckComparerByKind : IComparer<Duck> {
    public int Compare(Duck x, Duck y) {
        if (x.Kind < y.Kind)
            return -1;
        if (x.Kind > y.Kind)
            return 1;
        else
            return 0;
        }
}
```

这里比较了鸭子的Kind属性，所以会根据KindOfDuck enum的索引值对鸭子排序。

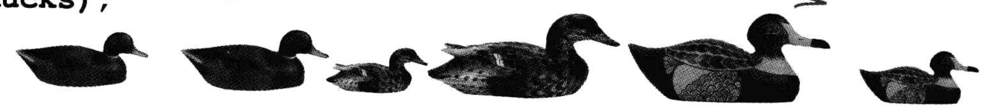
注意这里“大于”和“小于”有不同的含义。我们使用<和>来比较enum值，从而可以为鸭子排序。

这里是一个enum和List合作的例子。使用enum是为了提供数字，用于对列表排序。

```
DuckComparerByKind kindComparer = new DuckComparerByKind();
ducks.Sort(kindComparer);
PrintDucks(ducks);
```

Main()方法中的更多鸭子排序代码。

按鸭子种类排序……



**如果没有为Sort()提供一个IComparer<T>对象，它会使用一个可以对值类型排序或比较引用的默认对象。翻到附录“其他”中的第5项，可以学习更多有关比较对象的内容。**

## IComparer可以完成复杂的比较

专门创建一个单独的类对鸭子排序有一个好处，这样可以在这个类中建立更复杂的逻辑，而且可以增加一些成员来帮助确定列表要如何排序。

```
enum SortCriteria {
    SizeThenKind,
    KindThenSize,
}

class DuckComparer : IComparer<Duck> {
    public SortCriteria SortBy = SortCriteria.SizeThenKind;

    public int Compare(Duck x, Duck y) {
        if (SortBy == SortCriteria.SizeThenKind)
            if (x.Size > y.Size)
                return 1;
            else if (x.Size < y.Size)
                return -1;
            else
                if (x.Kind > y.Kind)
                    return 1;
                else if (x.Kind < y.Kind)
                    return -1;
                else
                    return 0;
        else
            if (x.Kind > y.Kind)
                return 1;
            else if (x.Kind < y.Kind)
                return -1;
            else
                if (x.Size > y.Size)
                    return 1;
                else if (x.Size < y.Size)
                    return -1;
                else
                    return 0;
    }
}

DuckComparer comparer = new DuckComparer();

comparer.SortBy = SortCriteria.KindThenSize;
ducks.Sort(comparer);
PrintDucks(ducks);

comparer.SortBy = SortCriteria.SizeThenKind;
ducks.Sort(comparer);
PrintDucks(ducks);
```

这个enum告诉对象用哪种方式对鸭子排序。

下面是一个比较鸭子的更复杂的类。Compare()方法的参数不变，不过现在它会查看公共SortBy字段来确定如何对鸭子排序。

这个if语句检查SortBy字段。如果SortBy设置为SizeThenKind，则首先按大小对鸭子排序，如果大小相同，再按鸭子的种类排序。

如果两个鸭子大小相同，这里不是简单地返回0，比较类会检查鸭子的种类，只有当两个鸭子大小相同而且是相同种类才会返回0。

如果SortBy并未设置为SizeThenKind，比较类首先按鸭子种类排序。如果两个鸭子种类相同，再来比较鸭子的大小。

这里展示了如何使用这个比较类对象。首先，像以往一样对它实例化。然后在调用ducks.Sort()之前设置对象的SortBy字段。现在通过修改对象中的一个字段就能改变鸭子排序的方式。把这个代码增加到Main()方法的最后。现在不仅可以排序，还可以重新排序很多次！

**Exercise**

创建5张随机的牌，然后对它们排序。

**1 创建代码，建立一组打乱的牌。**

创建一个新的控制台应用，向`Main()`方法增加代码创建5个随机的`Card`对象。创建各个对象之后，使用内置的`Console.WriteLine()`方法将牌名写至输出。在程序最后使用`Console.ReadKey()`，以防程序结束时窗口消失。

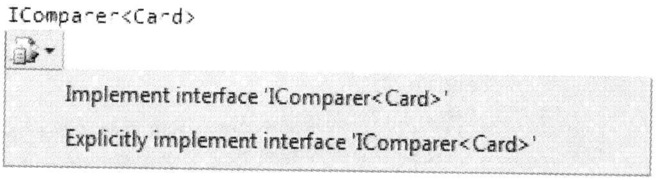
**2 创建一个实现了IComparer<Card>的类对这些牌排序。**

这里正好可以使用IDE为实现接口提供的快捷方法：

```
class CardComparer_byValue : IComparer<Card>
```

然后点击`IComparer`，并把鼠标停在I上。可以看到下方会出现一个框，点击这个框时，IDE会弹出一个“实现接口”窗口：

有时要弹出这个框有些困难，所以IDE提供了一个有用的快捷方法：可以按下ctrl，同时按下点号（.）。

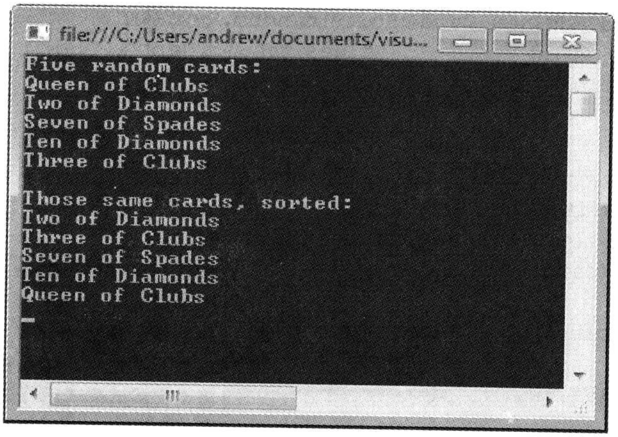



点击框中的“Implement interface IComparer<Card>”,IDE会自动填入要求实现的所有方法和属性。在这里，它会创建一个空的`Compare()`方法来比较两张牌（x和y）。编写这个方法，如果x大于y则返回1，如果x小于y返回-1，如果是相同的牌则返回0。在这里，要确保King(K)在Jack(J)后面，J在Four(4)后面，4又在Ace(A)后面。

**3 确保输出正确。**

点击按钮后输出窗口应当如下。

使用内置Console.WriteLine()方法时，它会向这个输出窗口增加一行。Console.ReadKey()会等待你按下一个按键后才终止程序。




IComparer对象需要按牌面大小排序，所以最小的牌就是列表中的第一张牌。




创建5张随机的牌，然后对它们排序。

这是扑克牌排序的"关键"所在，它使用了内置List.Sort()方法。Sort()取一个IComparer对象，它有一个方法：Compare()。这个实现取两张牌，首先比较它们的牌面大小，然后比较它们的花色。

```
class CardComparer_byValue : IComparer<Card> {
    public int Compare(Card x, Card y) {
        if (x.Value < y.Value) {
            return -1;
        }
        if (x.Value > y.Value) {
            return 1;
        }
        if (x.Suit < y.Suit) {
            return -1;
        }
        if (x.Suit > y.Suit) {
            return 1;
        }
        return 0;
    }
}
```

如果x牌面更大，返回1。如果x牌面较小，则返回-1。要记住，这两个return语句都会立即结束这个方法的执行。

只有当x和y的牌面大小相同时才会执行这些语句，这表示只有前两个return语句不执行时才会执行到这里。

如果这4个return语句都没有机会执行，说明两张牌肯定完全相同，因此返回0。

```
static void Main(string[] args)
{
    Random random = new Random();
    Console.WriteLine("Five random cards:");
    List<Card> cards = new List<Card>();
    for (int i = 0; i < 5; i++)
    {
        cards.Add(new Card((Suits)random.Next(4),
                           (Values)random.Next(1, 14)));
        Console.WriteLine(cards[i].Name);
    }

    Console.WriteLine();
    Console.WriteLine("Those same cards, sorted:");
    cards.Sort(new CardComparer_byValue());
    foreach (Card card in cards)
    {
        Console.WriteLine(card.Name);
    }
    Console.ReadKey();
}
```

这是一个泛型Card对象List，用来存储扑克牌。只要列表中有扑克牌，就能很容易地使用IComparer对它们排序。

**我们一直在使用Console.ReadKey()，以防止控制台应用完成时就立即退出。这对于学习来说很不错，不过如果你想编写真正的命令行应用时这就不太好了。如果使用Ctrl-F5启动程序，IDE将不在调试模式运行这个程序。程序完成时，会输出"Press any key to continue…"（请按任意键继续……），并等着你按键。不过这不会调试你的程序（因为它没有在调试模式下运行），所以你的断点和监视项都不起作用。**

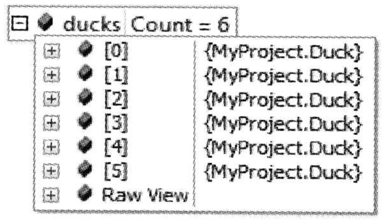
# 覆盖ToString()方法让对象描述自己

每个.NET对象都有一个名为ToString()的方法，这个方法可以把对象转换为一个字符串。默认地，它只是返回类名（MyProject.Duck）。这个方法是从Object继承的（应该记得，这是所有对象的基类）。这是一个非常有用的方法，而且确实得到了大量使用。例如，联接字符串的+操作符会自动调用一个对象的ToString()。另外Console.WriteLine()或String.Format()也会在传入对象时自动调用这个方法，如果你想把一个对象转换为一个字符串，这会很方便。

再来看前面的鸭子排序程序。在Main()方法中初始化列表之后的任意位置放置一个断点来调试这个程序。然后把鼠标停在某个鸭子变量上，这会在一个窗口中显示它的值。只要在调试工具中查看一个变量，而该变量中包含一个列表引用，就可以点击+按钮查看它的内容：




IDE在其监视窗口中显示一个对象时会调用ToString()方法。不过Duck从Object继承的ToString()方法只是返回它的类名。如果能让ToString()方法提供更多信息会非常有用。

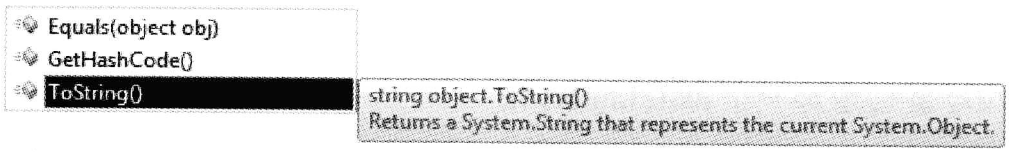



所以并不是向Console.WriteLine()、String.Format()等方法传入一个值，可以传入一个对象，它的ToString()方法会自动得到调用。对于像int和enum之类的值类型也同样适用！

嗯，这没有我们预想的那么有用。可以看到列表中有6个Duck对象（我们使用的命名空间是“MyProject”）。如果点击一个鸭子旁边的+按钮，可以看到它的Kind和Size值。不过如果能一次全部都能看到不是更容易吗？

很幸运，ToString()是Object（所有对象的基类）的一个虚方法。所以你要做的就是覆盖ToString()方法。覆盖后，你会立即在IDE的监视窗口中看到结果！打开你的Duck类，键入override开始增加一个新方法。一旦按下空格，IDE就会显示出你可以覆盖的方法：

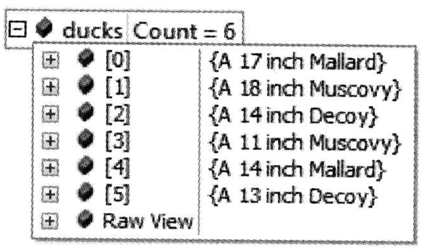
```
override
```




点击ToString()，告诉IDE增加一个新的ToString()方法。将内容替换如下所示：

```
public override string ToString()
{
    return "A " + Size + " inch " + Kind.ToString();
}
```

运行你的程序，再来看列表。现在IDE会显示出Duck的内容！




IDE的调试工具显示了一个对象，它调用这个对象的ToString()方法，向你显示出它的内容。

# 更新foreach循环，让Duck和Card能自行输出

你已经看到两个不同的程序例子，它们都循环处理一个对象列表，并调用Console.WriteLine()为每个对象分别向控制台输出一行。比如，下面这个foreach循环会输出一个List<Card>中的每张牌：

```
foreach (Card card in cards)
{
    Console.WriteLine(card.Name);
}
```

PrintDucks()方法对List中的Duck对象也做了类似的处理：

+操作符自动调用KindOfDuck enum的ToString()方法。

```
foreach (Duck duck in ducks)
{
    Console.WriteLine(duck.Size.ToString() + "-inch " + Kind);
}
```

这样处理对象相当常见。不过，既然Duck有一个ToString()方法，PrintDucks()方法就应该利用这一点：

```
public static void PrintDucks(List<Duck> ducks) {
    foreach (Duck duck in ducks) {
        Console.WriteLine(duck);
    }
Console.WriteLine("End of ducks!");
}
```

如果向Console.WriteLine()传入一个对象引用，它会自动调用这个对象的ToString()方法

将这个方法增加到Ducks程序中，并重新运行。它会打印同样的输出。现在如果你想向Duck对象增加一个属性，比如一个Gender属性，只需更新ToString()方法，使用它的所有方法［（包括PrintDucks()方法）］都会反映出这个变化。

## 向Card对象也增加一个ToString()方法

Card对象已经有一个Name属性，它会返回牌名：

```
public string Name
{
    get { return Value.ToString() + " of " + Suit.ToString(); }
}
```

仍然可以这样调用ToString()，不过现在你知道这种情况下这是不必要的，因为+会自动调用这个方法。

这正是这个ToString()方法要做的。所以向Card类增加一个ToString()方法：

```
public override string ToString()
{
    return Name;
}
```

ToString()非常有用，而不只是能够让对象在IDE中更容易标识。可以在后面几章中多注意一下，你会看到每个对象都有办法把自己转换为一个字符串是多么有用。正是因为这个原因，每个对象都有一个ToString()方法。

现在使用Card对象的程序将更易于调试。

# 写foreach循环时，就是在使用IEnumerable<T>

**foreach循环放大镜**

进入IDE，找到一个List<Duck>变量，使用智能提示查看它的GetEnumerator()方法。先键入“.GetEnumerator”，看看会显示什么：

集合初始化方法适用于任何IEnumerable<T>对象！

```
ducks.GetEnumerator
      GetEnumerator    List<Duck>.Enumerator List<Duck>.GetEnumerator()
                       Returns an enumerator that iterates through the System.Collections.Generic.List<T>.
```

增加一行代码，创建一个新的Duck对象数组：

```
Duck[] duckArray = new Duck[6];
```

然后，键入duckArray.GetEnumerator，数组也有一个GetEnumerator()方法。这是因为，所有列表和数组都实现了一个名为IEnumerable<T>的接口，它包含一个方法GetEnumerator()。这个方法会返回一个Enumerator对象。

正是这个Enumerator对象提供了循环机制，使你可以按顺序循环处理列表。下面是一个foreach循环，利用一个名为duck的变量循环处理List<Duck>：

```
foreach (Duck duck in ducks) {
    Console.WriteLine(duck);
}
```

**集合实现IEnumerable<T>时，就为你提供了一种方法可以编写一个循环按顺序循环处理其中的内容。**

这个循环在后台的实际工作如下：

```
IEnumerator<Duck> enumerator = ducks.GetEnumerator();
while (enumerator.MoveNext()) {
      Duck duck = enumerator.Current;
      Console.WriteLine(duck);
}
IDisposable disposable = enumerator as IDisposable;
if (disposable != null) disposable.Dispose();
```

理论上讲，应该多介绍一点，不过后面你就会了解……

(现在先不用担心最后两行代码。第9章还会了解IDisposable的更多内容)

这两个循环会打印同样的鸭子。你可以自己运行这两个程序看看，它们会有完全相同的输出。

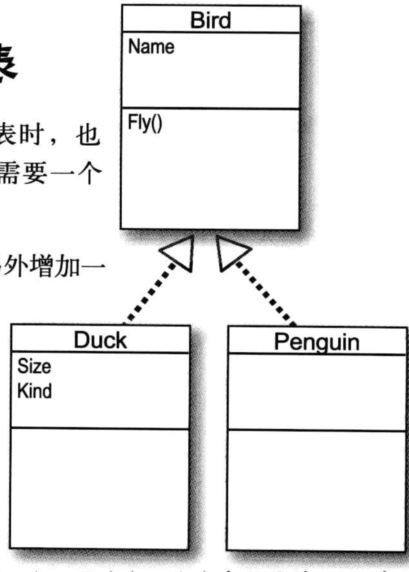
现在来说说到底发生什么。循环处理一个列表或数组（或者任何其他集合）时，如果列表中还有元素MoveNext()方法会返回true，或者如果enumerator已经到达列表末尾，这个方法就会返回false。Current属性是返回当前元素的一个引用。汇总在一起，你就得到了一个foreach循环！

试着改变Duck的ToString()，将Size属性增1。调试这个程序，把鼠标停在一个Duck上。然后再这样做一次。要记住，每次这样做时，IDE都会调用它的ToString()方法。

**如果ToString()方法会改变对象的某个字段，你认为foreach循环中会发生什么？**

# 可以使用IEnumerable向上强制转换整个列表

还记得如何把某个对象向上强制转换为它的超类吗？嗯，有一个对象列表时，也可以一次性向上强制转换整个列表。这称为协变（covariance），为此只需要一个IEnumerable<T>接口引用。




创建一个控制台应用，并增加一个基类，名为Bird(Duck将扩展这个类)，另外增加一个Penguin类。我们将使用ToString()方法以便查看究竟是哪个类。

```
class Bird {
    public string Name { get; set; }
    public void Fly() {
        Console.WriteLine("Flap, flap");
    }
    public override string ToString() {
        return "A bird named " + Name;
    }
}
```

```
class Penguin : Bird
{
    public void Fly() {
       Console.WriteLine("Penguins can't fly!");
    }
    public override string ToString() {
        return "A penguin named " + base.Name;
    }
}
```

这里有一个Bird类，另外有一个由它继承的Penguin类。把它们增加到一个新的控制台应用工程中，然后把你原来的Duck类也复制到这个工程中。只需改变Duck类的声明，使之扩展Bird。

```
class Duck : Bird, IComparable<Duck> {
    // The rest of the class is the same
}
```

这是Main()方法的前面几行代码，首先初始化列表，然后向上强制转换。

```
List<Duck> ducks = new List<Duck>() { // initialize your list as usual }
IEnumerable<Bird> upcastDucks = ducks;
```

复制前面用来初始化ducks列表的同一个集合初始化方法。

仔细看最后一行代码。这里得到List<Duck>的一个引用，把它赋给一个IEnumerable<Bird>接口变量。通过调试，你会看到它指出同一个对象。

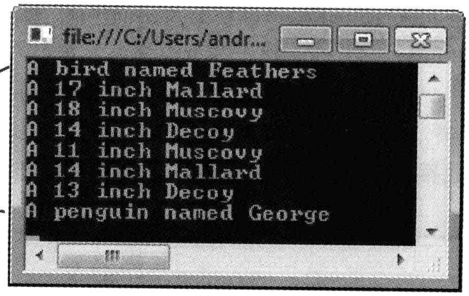
## 将各种鸟汇集到一个列表中

如果你想把一个对象集合增加到一个更通用的列表中，协变就非常有用。这里给出一个例子：如果你有一个Bird对象的列表，只需简单的一步就可以把你的Duck列表增加到这个列表中。下面的例子使用了List.AddRange()方法，这个方法可以用来把一个列表的内容增加到另一个列表中。

```
List<Bird> birds = new List<Bird>();

birds.Add(new Bird() { Name = "Feathers" });
birds.AddRange(upcastDucks);
birds.Add(new Penguin() { Name = "George" });

foreach (Bird bird in birds) {
    Console.WriteLine(bird);
}
```




一旦ducks向上强制转换为一个IEnumerable<Bird>，就可以把它们增加到一个Bird对象列表中了。

# 可以建立你自己的重载方法

前面已经使用了重载方法，甚至使用了作为内置.NET Framework类和对象一部分的一个重载构造函数，所以你已经了解了它们非常有用。如果能在你自己的类中建立重载方法不是很酷吗？没错，这是可以的，而且很容易！你要做的就是写两个或多个同名但参数不同的方法。

还可以使用一个using语句而不是改变命名空间。如果想更多地了解命名空间，花点时间翻到附录“其他”中的第2项。

**动手做！**

**1 创建一个新工程，并在其中增加Card类。**

这很容易做到，只需在Solution Explorer中右键点击工程，并从Add菜单选择“Existing Item”。IDE会建立这个类的一个副本，并把它增加到工程中。这个文件仍使用原工程的命名空间，所以在Card.cs文件最前面将命名空间一行改为你创建的新工程的名字。然后对Values和Suits enum做同样的处理。

如果没有这样做，只能通过明确指定命名空间来访问Card类（如oldnamespace.Card）。

**2 向card类增加一些新的重载方法。**

创建两个静态的DoesCardMatch()方法。第一个要检查一张牌的花色。第二张要检查它的牌面大小。只有当牌的花色或牌面大小一致时这两个方法才返回true。

```
public static bool DoesCardMatch(Card cardToCheck, Suits suit) {
   if (cardToCheck.Suit == suit) {
      return true;
   } else {
      return false;
   }
}
public static bool DoesCardMatch(Card cardToCheck, Values value) {
   if (cardToCheck.Value == value) {
      return true;
   } else {
      return false;
   }
}
```

重载方法并不要求是静态的，不过最好练习一下编写静态方法。

你已经见过重载。可以翻回到Kathleen的宴会规划程序解决方案（第6章253～256页），那里就向DinnerParty类增加了一个重载CalculateCost()方法。

**3 向窗体增加一个按钮来使用这些新方法。**

为按钮增加以下代码：

```
Card cardToCheck = new Card(Suits.Clubs, Values.Three);
bool doesItMatch = Card.DoesCardMatch(cardToCheck, Suits.Hearts);
MessageBox.Show(doesItMatch.ToString());
```

注意这里如何使用ToString()。这是因为MessageBox.Show()需要一个string而不是一个bool或对象。

一旦输入“DoesCardMatch(”，IDE会显示你确实已经建立了一个重载方法：

```
Card.DoesCardMatch(
▲ 1 of 2 ▼ bool Card.DoesCardMatch(Card cardToCheck, Suits suit)
```

花点时间尝试这两个方法来熟悉重载。

Exercise

使用List做些练习，建立一个类存储一副牌，另外建立一个窗体来使用这个类。

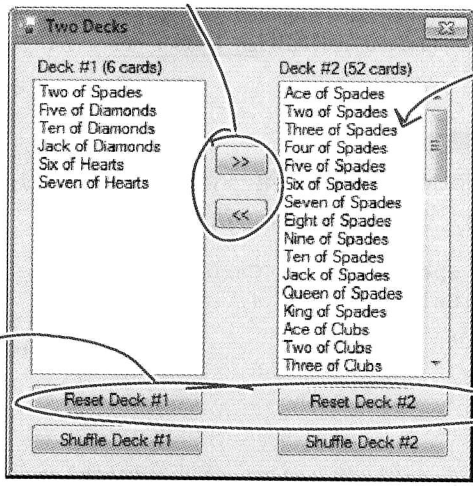
**1 建立一个窗体，从而能够在两副牌之间移牌。**

前面已经构建了一个Card类。现在创建一个类保存任意数量的牌，命名为`Deck`。一副牌实际有52张牌，但`Deck`类可以存放任意数量的牌，甚至可以一张牌也没有。

然后建立一个窗体，显示两个`Deck`对象的内容。第一次启动程序时，deck#1最多有10张随机的牌，deck #2是完整的一副52张牌，两副牌都先按花色再按牌面大小排序。可以使用两个Reset按钮将两副牌重置为其初始状态。窗体还有另外两个按钮（标签为“<<” 和“>>”），可以用来在这两副牌之间移牌。

这些按钮名为moveToDeck2（上面的按钮）和moveToDeck1（下面的按钮）。它们可以将牌从一副牌移到另一副牌。

**要记住，可以使用控件的Name属性为它指定一个名，这样能使代码更易读。双击这个按钮时，就会为其事件处理程序指定一个与按钮对应的方法名。**




使用两个ListBox控件来显示这两副牌。点击moveToDeck1按钮时，它将所选的牌从deck #2移到deck #1。

reset1和reset2按钮都首先调用ResetDeck()方法，然后调用RedrawDeck()方法。

这些按钮分别名为shuffle1和shuffle2。它们调用适当的Deck.Shuffle()方法，然后重新绘制这副牌。

除了6个按钮的事件处理程序，还需要为窗体增加两个方法。首先增加一个`ResetDeck()`方法，将一副牌重置为其初始状态。它取一个`int`参数：如果传入1，则重置第一个`Deck`对象，把它重新初始化为一副空牌，随机增加最多10张牌；如果传入2，则重置第2个`Deck`对象，使其中包含完整的52张牌。然后增加以下方法：

```
private void RedrawDeck(int DeckNumber) {
    if (DeckNumber == 1) {
        listBox1.Items.Clear();
        foreach (string cardName in deck1.GetCardNames())
            listBox1.Items.Add(cardName);
        label1.Text = "Deck #1 (" + deck1.Count + " cards)";
    } else {
        listBox2.Items.Clear();
        foreach (string cardName in deck2.GetCardNames())
            listBox2.Items.Add(cardName);
        label2.Text = "Deck #2 (" + deck2.Count + " cards)" ;
    }
}
```

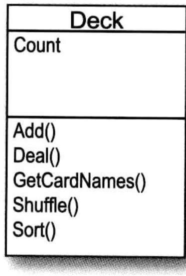
来看如何使用foreach循环将一副牌里的每张牌增加到列表框中。

RedrawDeck()方法会洗牌，从中随机抽牌，用两个Deck对象当前的内容来更新两个列表框控件。

**2** **构建Deck类。**

如果有类的声明而没有具体实现，这称为一个"骨架"。

以下是Deck类的骨架。我们已经为你填入了一些方法。现在需要完成这个类，写出Shuffle()和GetCardNames()方法，另外必须让Sort()方法正常工作。我们还增加了两个有用的重载构造函数：一个用于创建一副完整的52张牌，另一个构造函数取一个Card对象数组，把它们加载到这副牌中。




Deck把扑克牌存储在一个List中，不过它让这个List为私有，确保这个类得到很好的封装。

如果没有向构造函数传递参数，就创建一副完整的52张牌。

这个参数类型为IEnumerable<Card>，这就允许向这个构造函数传入任何集合，而不只是一个List<T>或数组。

这个重载构造函数取一个参数，即一个扑克牌数组，将加载这些牌作为初始的一副牌。

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();

    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }

    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }

    public int Count { get { return cards.Count; } }

    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }

    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }

    public void Shuffle() {
        // this method shuffles the cards by rearranging them in a random order
    }

    public IEnumerable<string> GetCardNames() {
        // this method returns a string array that contains each card's name
    }

    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

提示：ListBox控件的SelectedIndex属性与列表中这张牌的索引相同。可以把它直接传入Deal()方法。如果没有选择任何牌，它会小于0。在这种情况下，moveToDeck按钮什么也不做。

Deal方法从这副牌中出牌，它从这副牌中取出一个Card对象，并返回它的一个引用。可以传入0从最上面出牌，通过传入要出的牌的索引，也可以从这副牌中间出牌。

同样的，尽管GetCardNames()返回一个数组，但我们提供一个IEnumerable<string>。

你需要编写Shuffle()方法和GetCardNames()方法，并增加一个实现了IComparer的类，使Sort()方法能工作。另外，需要增加前面已经写的Card类。如果使用"Add Existing Item"特性来增加，不要忘记改变它的命名空间。

另一个提示：利用这个窗体可以很容易地测试你的Shuffle()方法。只要一直点击"Reset Deck #1"按钮直到得到的一副牌中包含3张牌。这样可以很容易地看出洗牌代码是否能正常工作。




创建一个存储一副牌的类，并建立一个窗体使用这个类。

```
class Deck {
    private List<Card> cards;
    private Random random = new Random();
    public Deck() {
        cards = new List<Card>();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                cards.Add(new Card((Suits)suit, (Values)value));
    }
    public Deck(IEnumerable<Card> initialCards) {
        cards = new List<Card>(initialCards);
    }
    public int Count { get { return cards.Count; } }
    public void Add(Card cardToAdd) {
        cards.Add(cardToAdd);
    }
    public Card Deal(int index) {
        Card CardToDeal = cards[index];
        cards.RemoveAt(index);
        return CardToDeal;
    }
    public void Shuffle() {
        List<Card> NewCards = new List<Card>();
        while (cards.Count > 0) {
            int CardToMove = random.Next(cards.Count);
            NewCards.Add(cards[CardToMove]);
            cards.RemoveAt(CardToMove);
        }
        cards = NewCards;
    }
    public IEnumerable<string> GetCardNames() {
        string[] CardNames = new string[cards.Count];
        for (int i = 0; i < cards.Count; i++)
            CardNames[i] = cards[i].Name;
        return CardNames;
    }
    public void Sort() {
        cards.Sort(new CardComparer_bySuit());
    }
}
```

这是创建一副完整的52张牌的构造函数。它使用了一个嵌套for循环。外循环迭代处理4个花色。这说明循环处理13个牌面大小的内循环会运行4次，对每个花色分别运行一次。

这是另一个构造函数。这个类有两个重载构造函数，分别有不同的参数。

Add和Deal方法很简单，它们使用了Cards列表的相应方法。Deal方法从列表中删除一张牌，Add方法在列表中增加一张牌。

Shuffle()方法创建List<Cards>的一个新实例，名为NewCards。然后从Cards字段随机地取牌，把它们插入NewCards中，直到Cards为空。完成后再重置Cards字段，指向这个新实例。现在再没有引用指向原来的老实例了，所以它会被垃圾回收。

GetCardNames()方法需要创建一个足够大的数组来存放所有牌名。这里使用了一个for循环，不过也可以使用foreach循环。

```
class CardComparer_bySuit : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Suit > y.Suit)
            return 1;
        if (x.Suit < y.Suit)
            return -1;
        if (x.Value > y.Value)
            return 1;
        if (x.Value < y.Value)
            return -1;
        return 0;
    }
}
```

按花色排序与按牌面大小排序很类似。唯一的区别在于，在这里会先比较花色，只有当花色相同时才会比较牌面大小。

这里没有使用if/else if，而是使用了一系列if语句。这是可以的，因为只有前面的if语句没有执行的情况下才会执行后面的if语句，否则，如果执行了前面的if语句会直接返回。

```
    Deck deck1;
    Deck deck2;
    Random random = new Random();

    public Form1() {
        InitializeComponent();
        ResetDeck(1);
        ResetDeck(2);
        RedrawDeck(1);
        RedrawDeck(2);
    }

    private void ResetDeck(int deckNumber) {
        if (deckNumber == 1) {
            int numberOfCards = random.Next(1, 11);
            deck1 = new Deck(new Card[] { });
            for (int i = 0; i < numberOfCards; i++)
                deck1.Add(new Card((Suits)random.Next(4),
                                   (Values)random.Next(1, 14)));
            deck1.Sort();
        } else
            deck2 = new Deck();
    }
```

窗体的构造函数需要重置两副牌，然后绘制出来。

要重置第1副牌（deck #1），这个方法首先使用random.Next()得出要向这副牌里增加多少张牌，然后创建一副空牌。接下来使用一个for循环按前面得出的牌数随机地增加一些牌。最后对这副牌排序。重置第2副牌（deck #2）很容易，只需创建Deck()的一个新实例。

前面的说明已经给出了RedrawDeck()方法。

未完，翻到下一页！

## Exercise Solution (续)

这是这个窗体的其余代码。

对控件命名，这样读代码时会更容易。如果只是命名为button1_Click、button2_Click等，你就无法知道在看哪个按钮的代码。

```
private void reset1_Click(object sender, EventArgs e) {
    ResetDeck(1);
    RedrawDeck(1);
}

private void reset2_Click(object sender, EventArgs e) {
    ResetDeck(2);
    RedrawDeck(2);
}

private void shuffle1_Click(object sender, EventArgs e) {
    deck1.Shuffle();
    RedrawDeck(1);
}

private void shuffle2_Click(object sender, EventArgs e) {
    deck2.Shuffle();
    RedrawDeck(2);
}

private void moveToDeck1_Click(object sender, EventArgs e) {
    if (listBox2.SelectedIndex >= 0)
        if (deck2.Count > 0) {
            deck1.Add(deck2.Deal(listBox2.SelectedIndex));
        }
    RedrawDeck(1);
    RedrawDeck(2);
}

private void moveToDeck2_Click(object sender, EventArgs e) {
    if (listBox1.SelectedIndex >= 0)
        if (deck1.Count > 0)
            deck2.Add(deck1.Deal(listBox1.SelectedIndex));
    RedrawDeck(1);
    RedrawDeck(2);
}
}
```

这些按钮很简单，首先重置或洗牌，再重新绘制。

可以使用ListBox控件的SelectedIndex属性来确定用户选择了哪张牌，并把它从一副牌移到另一副牌（如果小于0，说明没有选择任何牌，这个按钮什么也不做）。移完牌后，两副牌都要重绘。

# 使用字典存储键和值

列表就像长长的一页纸，里面满是名字。但是，如果你希望对应每个名字有一个地址该怎么做呢？或者对于车库清单里的每辆车，希望得到有关的详细信息又该如何实现？对于这些情况，你需要一个字典（dictionary）。利用字典，可以将一个特殊的值（键）与一组数据（值）关联。还有一点要注意：一个特定键在字典中只能出现一次。

这就是键（可以想见）要通过它在字典中查找定义。

**dic·tion·ar·y**

这种书把一种语言的单词按字母顺序列出来，并给出各个单词的含义。

这就是值。这是与一个特定键关联的数据。

在C#中要如下声明`Dictionary`：

```
Dictionary <Tkey, TValue> kv = new Dictionary <TKey, TValue>();
```

这些与List<T>很类似。<T>表示放在这里的一种类型。所以可以为键声明一种类型，而为值声明另一种类型。

这些表示类型。尖括号里的第一项总是键，第二项总是值。

这是`Dictionary`的具体使用：

```
private void button1_Click(object sender, EventArgs e)
{
    Dictionary<string, string> wordDefinition =
        new Dictionary<string, string>();

    wordDefinition.Add ("Dictionary", "A book that lists the words of a "
                    + "language in alphabetical order and gives their meaning");
    wordDefinition.Add ("Key", "A thing that provides a means of gaining access to "
                    + "our understanding something.");
    wordDefinition.Add ("Value", "A magnitude, quantity, or number.");

    if (wordDefinition.ContainsKey("Key")){
        MessageBox.Show(wordDefinition["Key"]);
    }

}
```

这个字典的键为string值，另外值也是string值。这就像一个真正的字典：前面是词条，后面是定义。

要通过Add()方法向字典增加键和值。

Add()取一个键，然后是值。

ContainsKey()会告诉你一个键是否在字典中。是不是很方便？

这样可以得到对应一个键的值。这有些类似于数组索引（得到这个索引对应的键的值）。

# 字典功能通览

字典与列表非常类似。这两种类型都很灵活，允许处理多种数据类型，而且提供了大量内置的功能。以下是一些基本的Dictionary方法：

★ **增加一项。**
将一个键和一个值传入Add()方法，可以向字典增加一项。

```
Dictionary<string, string> myDictionary = new Dictionary<string, string>();
myDictionary.Add("some key", "some value");
```

★ **使用键查找一个值。**
使用字典时最重要的就是查找值，这很有道理，毕竟，在字典中存储这些值的目的就是为了能使用唯一的键把它们找出来。对于这个Dictionary<string, string>，要使用一个string键来查找值，这会返回一个string。

```
string lookupValue = myDictionary["some key"];
```

★ **删除一项。**
类似于List，可以使用Remove()方法从字典删除一项。只需向Remove方法传入Key值，这一项的键和值都将删除。

```
myDictionary.Remove("some key");
```

一个字典中的键是唯一的：任何键只能出现一次。值可以出现任意多次，两个键可以有相同的值。因此，查找或删除一个键时，Dictionary能知道究竟要删除什么。

★ **得到键列表。**
使用Keys属性并通过一个foreach循环迭代处理，可以得到Dictionary中所有键的一个列表。通常可以如下使用：

```
foreach (string key in myDictionary.Keys) { ... };
```

Keys是字典对象的一个属性。这个特定字典的键是string类型，所以Keys是一个string集合。

★ **统计字典中的键值对数。**
Count属性会返回字典中键值对的数目：

```
int howMany = myDictionary.Count;
```

## 键和值可以是不同的类型

字典确实是个多面手，可以存放几乎任何东西，从字符串到数字，甚至对象都可以存放。下面是一个字典的例子，这里存储一个整数作为键，另外存储duck对象作为值。

为对象指定唯一的ID号时通常会看到将整数映射到对象的字典。

```
Dictionary<int, Duck> duckDictionary = new Dictionary<int, Duck>();
duckDictionary.Add(376, new Duck()
                    { Kind = KindOfDuck.Mallard, Size = 15 });
```

# 构建一个使用字典的程序

所有纽约棒球迷都会喜欢下面这个简单的小程序。一个明星球员退役时，球队也会将这个球员的球衣退役。下面来建立一个程序，查看那些出名号码的球衣是谁穿的，还有这些号码是什么时间退役的。下面是跟踪球衣号码的类：

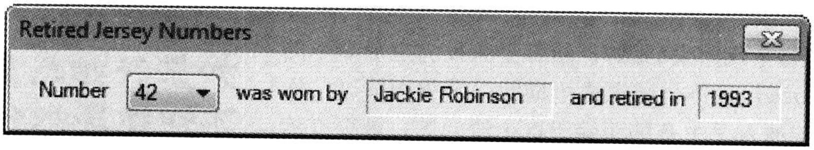
**动手做！**

Yogi Berra 是一个球队的8号球员，Cal Ripken Jr.是另一个球队的8号球员。但是在一个字典中一个键只能映射一个值，所以这里只包含一个球队的号码。你能想出一种办法来存储多个球队的退役号码吗？

```
class JerseyNumber {
    public string Player { get; private set; }
    public int YearRetired { get; private set; }

    public JerseyNumber(string player, int numberRetired) {
        Player = player;
        YearRetired = numberRetired;
    }
}
```

以下是窗体：




这里是窗体的所有代码：

```
public partial class Form1 : Form {
  Dictionary<int, JerseyNumber> retiredNumbers = new Dictionary<int, JerseyNumber>() {
      {3, new JerseyNumber(“Babe Ruth”, 1948)},
      {4, new JerseyNumber(“Lou Gehrig”, 1939)},
      {5, new JerseyNumber(“Joe DiMaggio”, 1952)},
      {7, new JerseyNumber(“Mickey Mantle”, 1969)},
      {8, new JerseyNumber(“Yogi Berra”, 1972)},
      {10, new JerseyNumber(“Phil Rizzuto”, 1985)},
      {23, new JerseyNumber(“Don Mattingly”, 1997)},
      {42, new JerseyNumber(“Jackie Robinson”, 1993)},
      {44, new JerseyNumber(“Reggie Jackson”, 1993)},
  };

  public Form1() {
      InitializeComponent();
      foreach (int key in retiredNumbers.Keys) {
          number.Items.Add(key);
      }
  }

  private void number_SelectedIndexChanged(object sender, EventArgs e) {
      JerseyNumber jerseyNumber = retiredNumbers[(int)number.SelectedItem] as JerseyNumber;
      nameLabel.Text = jerseyNumber.Player;
      yearLabel.Text = jerseyNumber.YearRetired.ToString();
  }
}
```

使用一个集合初始化方法用JerseyNumber对象填充Dictionary。

从字典将各个键增加到ComboBox的Items集合。

ComboBox的SelectedItem属性是一个对象。由于字典的键是一个int，所以在字典中进行查找之前先要对它完成强制转换。

使用ComboBox的SelectedIndexChanged事件，用从字典获取的JerseyNumber对象的值更新窗体上的两个标签。

## Long Exercise

构建一个钓鱼（Go Fish!）游戏，可以和计算机玩这个游戏。

**这个练习稍有些不同……**

你之所以学C#，很可能是因为想找这样一份工作，做一个专业的开发人员。所以我们按照专业的模式来布置这个练习。作为一个程序员参与团队工作时，你通常不会从头到尾构建完整的程序。相反，你只是构建一个更大程序中的某一部分。所以，下面将留一个问题，其中一些部分已经完成。窗体的代码在第3步给出。只需直接键入这些代码，看起来似乎起点很高，因为已经做了一些工作，但这些代码也带来了限制，这说明你的类必须与这些代码合作。而这可能是一个很大的挑战！

**1 先从规范开始。**

所有专业软件工程都从一个规范开始，这个工程也不例外。你要构建一个经典的扑克牌游戏钓鱼（Go Fish!）。不同的人玩这个游戏的规则可能稍有差别，所以下面先扼要介绍你要使用的规则：

- 游戏从52张牌开始。给每个玩家发5张牌。给每个人发完之后剩下的牌称为未发完的牌（stock）。每个玩家挨个询问一个牌面值（“你有没有7？”），如果其他人手中有牌面等于这个值的牌，就要交给这个玩家。如果所有人都没有这样的牌，玩家就必须“钓鱼”，从没发完的牌里取一张牌。
- 这个游戏的最终目标是凑成套牌（book），一套牌就是有相同牌面的全部4张不同花色的牌。到游戏最后，哪个玩家拿到的套数最多，谁就是赢家。一个玩家一旦收集到一套牌，就会把它面朝上放在桌子上，这样所有其他玩家就能看到每个人都拿到了哪些套牌。
- 一个玩家把一套牌放在桌子上时，他的牌可能出光。如果是这样，他必须从没发完的牌中再抽5张牌。如果剩下的牌不足5张，就把所有剩下的牌都取光。一旦所有牌都取光，游戏就结束了。然后根据谁的套数最多选出赢家。
- 对于这个计算机版的钓鱼游戏，有两个计算机玩家和一个人类玩家。每一轮从人类玩家开始，他先选择手中的一张牌，这张牌会一直显示。为此，玩家要选择一张牌，并提示他要问牌了。然后两个计算机玩家也会问牌。游戏中将显示每一轮的结果。这个过程会重复，直到出现赢家。
- 这个游戏要负责所有牌的交换，还要自动取出成套的牌。一旦出现赢家，游戏就结束。这个游戏会显示赢家的名字（或者可能有多个赢家，因为有可能有平局）。玩家必须重启程序才能开始一个新游戏，除此以外没有别的办法。

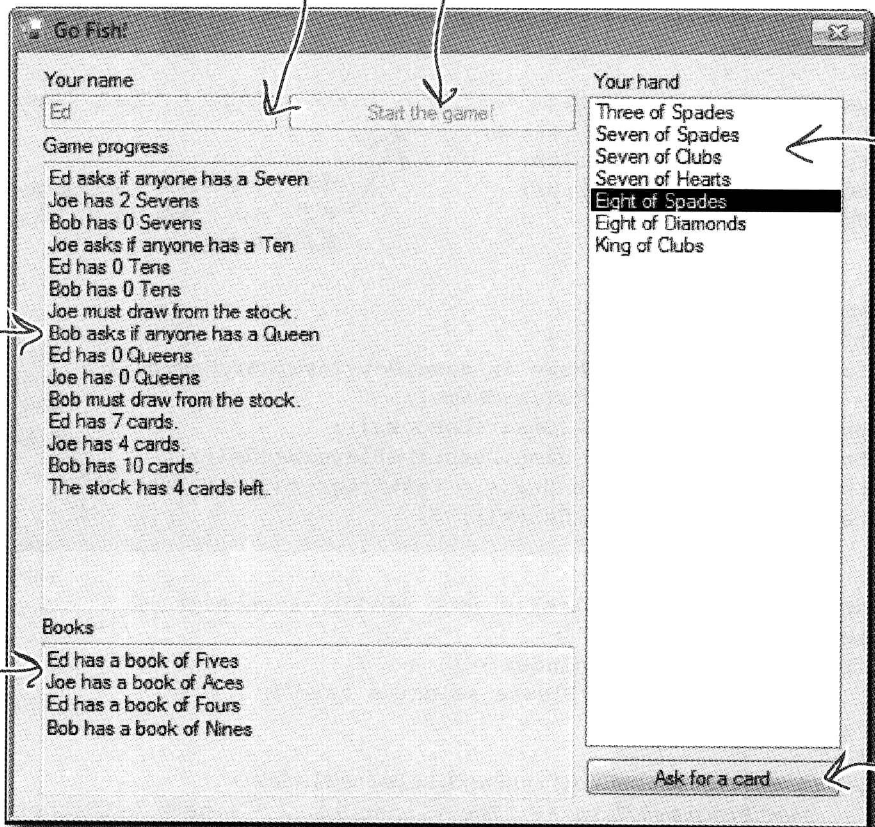
如果开始之前不知道要做什么，你怎么能知道什么时候才算结束呢？正是因为这个原因，大多数专业软件工程都从一个规范开始，指出要构建一个什么样的程序。

**2** **创建窗体。**

为这个钓鱼游戏创建窗体。应该有一个ListBox控件表示玩家手中的牌，两个TextBox控件表示游戏的进度，还有一个按钮允许玩家问牌。玩这个游戏时，用户要选择手中的一张牌，并点击这个按钮询问计算机玩家是否有牌面大小相同的牌。

这个TextBox控件的Name属性应当设置为textName。在这个切屏图中，它是禁用的，但是程序开始时会启用。

将这个按钮的Name属性设置为buttonStart。在这个切屏图中它是禁用的，但是游戏一旦开始就会启用。




玩家当前这手牌显示在一个ListBox控件中，名为listHand。可以使用Name属性来设置这个控件名。

这些是TextBox控件，分别名为textProgress和textBooks。

设置两个TextBox控件的ReadOnly属性为True，使它们成为只读文本框，另外设置Multiline属性为true。

设置这个按钮的Name属性为buttonAsk，并设置其Enabled属性为False。这会将其禁用，这说明不能按这个按钮。游戏一旦开始，窗体就会启用这个按钮。

未完，翻到下一页！

## Long Exercise（续）

**3** **以下是窗体是代码。**

把你看到的代码原样键入。余下的代码由你自己编写，你写的代码不能与这个代码冲突。

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
    }

    private Game game;

    private void buttonStart_Click(object sender, EventArgs e) {
        if (String.IsNullOrEmpty(textName.Text)){
            MessageBox.Show("Please enter your name", "Can't start the game yet");
            return;
        }
        game = new Game(textName.Text, new List<string> { "Joe", "Bob" }, textProgress);
        buttonStart.Enabled = false;
        textName.Enabled = false;
        buttonAsk.Enabled = true;
        UpdateForm();
    }

    private void UpdateForm() {
        listHand.Items.Clear();
        foreach (String cardName in game.GetPlayerCardNames())
            listHand.Items.Add(cardName);
        textBooks.Text = game.DescribeBooks();
        textProgress.Text += game.DescribePlayerHands();
        textProgress.SelectionStart = textProgress.Text.Length;
        textProgress.ScrollToCaret();
    }

    private void buttonAsk_Click(object sender, EventArgs e) {
        textProgress.Text = "";
        if (listHand.SelectedIndex < 0) {
            MessageBox.Show("Please select a card");
            return;
        }
        if (game.PlayOneRound(listHand.SelectedIndex)) {
            textProgress.Text += "The winner is... " + game.GetWinnerName();
            textBooks.Text = game.DescribeBooks();
            buttonAsk.Enabled = false;
        } else
            UpdateForm();
    }
}
```

窗体只与这个类交互。它将运行整个游戏。

Enabled属性可以启用或禁用窗体上的一个控件。

开始一个新游戏时，会创建Game类的一个新实例，启用“Ask”按钮，禁用“Start Game”按钮，然后重新绘制窗体。

这个方法可以清空并重新填充ListBox（其中包含玩家当前这手牌），然后更新文本框。

这样使用SelectionStart和ScrollToCaret()会把文本框滚动到最后，因此，如果一次要显示的文本太多，它会滚动到最下面。

SelectionStart行将闪动的文本框光标移到最后，一旦移动光标，ScrollToCaret()方法将把文本框滚动到光标所在位置。

玩家选出一张牌，并点击“Ask”按钮，查看其他玩家是否有同样牌面的牌。Game类使用PlayOneRound()方法玩一轮。

**4** **你还需要以下代码。**

你还需要先前为Card类、Suits和Values enum、Deck类和CardComparer _ byValue类写的代码。此外，还需要为Deck类增加几个方法…… 另外需要真正理解这些代码，才能很好地使用。

```
public Card Peek(int cardNumber) {
    return cards[cardNumber];
}
```

Peek()方法允许查看一副牌中的某一张牌而不用出牌。

```
public Card Deal() {
    return Deal(0);
}
```

这里重载了Deal()，这样更易读。如果没有传入任何参数，就从这副牌最上面出牌。

```
public bool ContainsValue(Values value) {
    foreach (Card card in cards)
        if (card.Value == value)
            return true;
    return false;
}
```

ContainsValue()方法在整副牌中搜索有指定牌面大小的牌，如果找到则返回true。你能猜出在钓鱼游戏中如何使用这个方法吗？

```
public Deck PullOutValues(Values value) {
    Deck deckToReturn = new Deck(new Card[] { });
    for (int i = cards.Count - 1; i >= 0; i--)
        if (cards[i].Value == value)
            deckToReturn.Add(Deal(i));
    return deckToReturn;
}
```

构建代码从一副牌中得到一套牌时，要使用PullOutValues()方法。它查看与某个牌面大小匹配的牌，把它们从这副牌中取出，并返回一副新牌，其中包含取出的这些牌。

```
public bool HasBook(Values value) {
    int NumberOfCards = 0;
    foreach (Card card in cards)
        if (card.Value == value)
            NumberOfCards++;
    if (NumberOfCards == 4)
        return true;
    else
        return false;
}
```

HasBook()方法检查一副牌，查看是否包含一套4张牌（这套牌的牌面作为参数传入）。如果这副牌中有一套牌，则返回true，否则返回false。

```
public void SortByValue() {
    cards.Sort(new CardComparer_byValue());
}
```

SortByValue()方法使用Comparer_byValue类对这副牌排序。

→ 未完，翻到下一页！

## Long Exercise (续)

5 **下面是最难的部分：构建Player类。**

对应游戏中的三个玩家分别有一个Player类的实例。它们由buttonStart按钮的事件处理程序创建。

仔细查看各个注释，这些注释指出了这些方法要做什么。你的任务就是具体填写这些方法。

```
class Player
{
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private TextBox textBoxOnForm;
    public Player(String name, Random random, TextBox textBoxOnForm) {
       // The constructor for the Player class initializes four private fields, and then
       // adds a line to the TextBox control on the form that says, "Joe has just
       // joined the game" - but use the name in the private field, and don't forget to
       // add a line break at the end of every line you add to the TextBox.
    }
    public IEnumerable<Values> PullOutBooks() {  } // see the facing page for the code
    public Values GetRandomValue() {
       // This method gets a random value—but it has to be a value that's in the deck!
    }
    public Deck DoYouHaveAny(Values value) {
       // This is where an opponent asks if I have any cards of a certain value
       // Use Deck.PullOutValues() to pull out the values. Add a line to the TextBox
       // that says, "Joe has 3 sixes" - use the new Card.Plural() static method
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock) {
       // Here's an overloaded version of AskForACard() - choose a random value
       // from the deck using GetRandomValue() and ask for it using AskForACard()
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
       // Ask the other players for a value. First add a line to the TextBox: "Joe asks
       // if anyone has a Queen". Then go through the list of players that was passed in
       // as a parameter and ask each player if he has any of the value (using his
       // DoYouHaveAny() method). He'll pass you a deck of cards - add them to my deck.
       // Keep track of how many cards were added. If there weren't any, you'll need
       // to deal yourself a card from the stock (which was also passed as a parameter),
       // and you'll have to add a line to the TextBox: "Joe had to draw from the stock"
    }
    // Here's a property and a few short methods that were already written for you
    public int CardCount { get { return cards.Count; } }
    public void TakeCard(Card card) { cards.Add(card); }
    public IEnumerable<string> GetCardNames() { return cards.GetCardNames(); }
    public Card Peek(int cardNumber) { return cards.Peek(cardNumber); }
    public void SortHand() { cards.SortByValue(); }
}
```

将会用到增加到Deck类的Peek()方法。利用这个方法，通过给定某张牌的索引号，程序可以在一副牌中查看这张牌，但是不同于Deal()，这个方法不会将这张牌删除。

```
public IEnumerable<Values> PullOutBooks() {
    List<Values> books = new List<Values>();
    for (int i = 1; i <= 13; i++) {
        Values value = (Values)i;
        int howMany = 0;
        for (int card = 0; card < cards.Count; card++)
            if (cards.Peek(card).Value == value)
                howMany++;
        if (howMany == 4) {
            books.Add(value);
            for (int card = cards.Count - 1; card >= 0; card--)
                cards.Deal(card);
        }
    }
    return books;
}
```

必须建立AskForACard()方法的两个重载版本。第一个由对手在要牌时使用，检查他们手中的牌，找出所要的牌。第二个重载方法由玩家在要牌时使用。这两个方法都会询问另外的每一个玩家（包括计算机和人类玩家），查找有指定牌面大小的牌。

**6** **需要向Card类增加这个方法。**

这是一个静态方法，取一个牌面大小，返回其复数形式，也就是说，如果输入10则返回“Tens”，输入6将返回“Sixes”（在后面加“es”）。由于这是一个静态方法，可以用类名直接调用，即Card.Plural()，而不需要从一个实例调用。

```
public partial class Card {
    public static string Plural(Values value) {
        if (value == Values.Six)
            return "Sixes";
        else
            return value.ToString() + "s";
    }
}
```

我们使用了一个部分类向Card增加这个静态方法，以便你看到做了什么。不过并不要求非得使用一个部分类。如果你愿意，也可以把它直接增加到现在的Card类中。

就快完了，继续翻下一页！

## Long Exercise（续）

**7** **余下的任务：构建Game类。**

窗体会维护Game的一个实例。它要管理游戏的运行。请仔细看窗体中是如何使用这个类的。

Player和Game类都使用窗体上多行TextBox的一个引用为用户输出一个消息以便阅读。一定要在这些文件最前面加上“using System.Windows.Forms;”。

```
class Game {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;
    private TextBox textBoxOnForm;

    public Game(string playerName, IEnumerable<string> opponentNames, TextBox textBoxOnForm) {
        Random random = new Random();
        this.textBoxOnForm = textBoxOnForm;
        players = new List<Player>();
        players.Add(new Player(playerName, random, textBoxOnForm));
        foreach (string player in opponentNames)
            players.Add(new Player(player, random, textBoxOnForm));
        books = new Dictionary<Values, Player>();
        stock = new Deck();
        Deal();
        players[0].SortHand();
    }

    private void Deal() {
        // This is where the game starts - this method's only called at the beginning
        // of the game. Shuffle the stock, deal five cards to each player, then use a
        // foreach loop to call each player's PullOutBooks() method.
    }

    public bool PlayOneRound(int selectedPlayerCard) {
        // Play one round of the game. The parameter is the card the player selected
        // from his hand - get its value. Then go through all of the players and call
        // each one's AskForACard() methods, starting with the human player (who's
        // at index zero in the Players list - make sure he asks for the selected
        // card's value). Then call PullOutBooks() - if it returns true, then the
        // player ran out of cards and needs to draw a new hand. After all the players
        // have gone, sort the human player's hand (so it looks nice in the form).
        // Then check the stock to see if it's out of cards. If it is, reset the
        // TextBox on the form to say, "The stock is out of cards. Game over!" and return
        // true. Otherwise, the game isn't over yet, so return false.
    }

    public bool PullOutBooks(Player player) {
        // Pull out a player's books. Return true if the player ran out of cards, otherwise
        // return false. Each book is added to the Books dictionary. A player runs out of
        // cards when he's used all of his cards to make books—and he wins the game.
    }

    public string DescribeBooks() {
        // Return a long string that describes everyone's books by looking at the Books
        // dictionary: "Joe has a book of sixes. (line break) Ed has a book of Aces."
    }
```

**在公共类成员中使用IEnumerable<T>是一种很好的办法，可以让类更灵活，如果你的代码需要重用就要考虑这一点。现在别人就可以用string[]、List<string>，或者任何其他类型来实例化Game类。**

这对于封装也很有好处。如果公布一个IEnumerable<T>而不是其他类型（比如说List<T>），你就不会无意中编写代码对它进行修改。

这里给出编写GetWinnerName()方法的一个提示：要在这个方法最前面创建一个新的Dictionary<string, int>，名为winners。winners字典允许你使用各个玩家的名字来查找他在游戏中得到的套牌数。首先使用一个foreach循环对玩家得到的套牌循环处理，并建立字典。然后使用另一个foreach循环查找与玩家关联的最大套牌数。不过可能会出现平局，有可能不只一个玩家拿到最多的套牌！所以还需要另一个foreach循环在winners的所有玩家中查找哪些玩家拿到的套牌数与第二个循环中得到的最大套牌数相同，并建立一个字符串指出哪些人是赢家。

```
public string GetWinnerName() {
        // This method is called at the end of the game. It uses its own dictionary
        // (Dictionary<string, int> winners) to keep track of how many books each player
        // ended up with in the books dictionary. First it uses a foreach loop
        // on books.Keys -- foreach (Values value in books.Keys) -- to populate
        // its winners dictionary with the number of books each player ended up with.
        // Then it loops through that dictionary to find the largest number of books
        // any winner has. And finally it makes one last pass through winners to come
        // up with a list of winners in a string ("Joe and Ed"). If there's one winner,
        // it returns a string like this: "Ed with 3 books". Otherwise it returns a
        // string like this: "A tie between Joe and Bob with 2 books."
}

// Here are a couple of short methods that were already written for you:

public IEnumerable<string> GetPlayerCardNames() {
    return players[0].GetCardNames();
}

public string DescribePlayerHands() {
    string description = "";
    for (int i = 0; i < players.Count; i++) {
        description += players[i].Name + " has " + players[i].CardCount;
        if (players[i].CardCount == 1)
            description += " card." + Environment.NewLine;
        else
            description += " cards." + Environment.NewLine;
    }
    description += "The stock has " + stock.Count + " cards left.";
    return description;
}
```

进入监视窗口，键入(int)'\r'把字符\r强制转换为一个数。这会转换为13。'\n'会转换为10。每个字符都会转换为相应的一个唯一数字，称为这个字符的Unicode值。有关内容将在下一章介绍。

## 使用Envrionment.NewLine增加换行符

这本书里一直在用\n向消息框增加换行符。.NET还提供了一个方便的常量来增加换行符：Environment.NewLine。其中总包含常量值"\r\n"。如果具体查看Windows格式文本文件中的字符，可以看到每行最后都有两个字符：'\r'和'\n'。另外一些操作系统（如Unix）只使用一个'\n'指示各行末尾。MessageBox.Show()方法非常聪明，它能自动地把\n'字符转换为换行符，不过如果使用Environment.NewLine而不是转义字符，你的代码会更易读。另外，使用Console.WriteLine()时向各行末尾追加的就是Environment.NewLine。

# Long Exercise Solution

以下是Game类中已填好的方法。

第一次游戏开始时调用Deal()方法，它先洗牌，然后向每个玩家发5张牌。如果玩家手中的牌恰好出现了一套牌，就把这套牌取出。

```
private void Deal() {
    stock.Shuffle();
    for (int i = 0; i < 5; i++)
        foreach (Player player in players)
            player.TakeCard(stock.Deal());
    foreach (Player player in players)
        PullOutBooks(player);
}

public bool PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            textBoxOnForm.Text += players[i].Name
                       + " drew a new hand" + Environment.NewLine;
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        players[0].SortHand();
        if (stock.Count == 0) {
            textBoxOnForm.Text =
                "The stock is out of cards. Game over!" + Environment.NewLine;
            return true;
        }
    }
    return false;
}
```

玩家或对手要牌时，游戏会取出已凑成的所有套牌。如果一个玩家手中的牌全部取完，再从没发完的牌中取5张牌成为一手新牌。

一旦玩家点击“Ask for a card”按钮，游戏就根据这张牌调用AskForACard()。然后对各个对手调用AskForACard()。

一轮结束后，游戏对玩家手中的牌排序，确保在窗体上按顺序显示。然后查看游戏是否结束。如果结束，则PlayOneRound()返回true。

```
public bool PullOutBooks(Player player)
{
    IEnumerable<Values> booksPulled = player.PullOutBooks();
    foreach (Values value in booksPulled)
        books.Add(value, player);
    if (player.CardCount == 0)
        return true;
    return false;
}
```

PullOutBooks()检查一个玩家的牌，查看是否有4张牌的牌面大小相同。如果是，就把这套牌增加到他的套牌字典中。如果没有牌剩下，则返回true。

窗体需要显示套牌的一个列表，所以使用DescribeBooks()将玩家的套牌字典转换为字符串。

```
public string DescribeBooks() {
    string whoHasWhichBooks = "";
    foreach (Values value in books.Keys)
        whoHasWhichBooks += books[value].Name + " has a book of "
            + Card.Plural(value) + Environment.NewLine;
    return whoHasWhichBooks;
}

public string GetWinnerName() {
    Dictionary<string, int> winners = new Dictionary<string, int>();
    foreach (Values value in books.Keys) {
        string name = books[value].Name;
        if (winners.ContainsKey(name))
            winners[name]++;
        else
            winners.Add(name, 1);
    }
    int mostBooks = 0;
    foreach (string name in winners.Keys)
        if (winners[name] > mostBooks)
            mostBooks = winners[name];
    bool tie = false;
    string winnerList = "";
    foreach (string name in winners.Keys)
        if (winners[name] == mostBooks)
        {
            if (!String.IsNullOrEmpty(winnerList))
            {
                winnerList += " and ";
                tie = true;
            }
            winnerList += name;
        }
    winnerList += " with " + mostBooks + " books";
    if (tie)
        return "A tie between " + winnerList;
    else
        return winnerList;
}
```

一旦选出了最后一张牌，游戏要确定谁是赢家。GetWinnerName()就要做这个工作，为此它要使用一个字典（名为winners）。每个玩家的名字是字典中的一个键，值是这个玩家在游戏中得到的套牌数。

接下来游戏检查这个字典，得出套数最多的玩家拿到了多少套。它把这个值放在一个名为mostBooks的变量中。

既然知道了哪个玩家拿到的套数最多，这个方法就能提供一个字符串，列出这个赢家（或者多个赢家）。

未完，翻到下一页！

# Long Exercise Solution（续）

以下是已填好的Player类方法。

```
public Player(String name, Random random, TextBox textBoxOnForm) {
    this.name = name;
    this.random = random;
    this.textBoxOnForm = textBoxOnForm;
    this.cards = new Deck( new Card[] {} );
    textBoxOnForm.Text += name +
              " has just joined the game" + Environment.NewLine;
}
```

这是Player类的构造函数。它设置私有字段，并为progress文本框增加一行指出谁加入这个游戏。

```
public Values GetRandomValue() {
    Card randomCard = cards.Peek(random.Next(cards.Count));
    return randomCard.Value;
}
```

GetRandomValue()方法使用Peek()来查看玩家手中一张随机的牌。

```
public Deck DoYouHaveAny(Values value) {
    Deck cardsIHave = cards.PullOutValues(value);
    textBoxOnForm.Text += Name + " has " + cardsIHave.Count + " "
           + Card.Plural(value) + Environment.NewLine;
    return cardsIHave;
}
```

DoYouHaveAny()使用PullOutValues()方法取出并返回与参数匹配的所有牌。

```
public void AskForACard(List<Player> players, int myIndex, Deck stock) {
    Values randomValue = GetRandomValue();
    AskForACard(players, myIndex, stock, randomValue);
}
```

有两个重载的AskForACard()方法。这个方法由对手使用，它从这手牌中得到一张随机的牌，并调用另一个AskorACard()重载方法。

**附加小练习：你能想办法改进Player类中的封装和设计吗？将这两个方法中的List<Player>替换为IEnumerable<Player>而不改变软件的工作方式。翻到附录“其他”中的第7项，其中给出了一个有用的工具可以提供帮助。**

```
public void AskForACard(List<Player> players, int myIndex,
      Deck stock, Values value) {
    textBoxOnForm.Text += Name + " asks if anyone has a "
                              + value  + Environment.NewLine;
    int totalCardsGiven = 0;
    for (int i = 0; i < players.Count; i++) {
        if (i != myIndex) {
            Player player = players[i];
            Deck CardsGiven = player.DoYouHaveAny(value);
            totalCardsGiven += CardsGiven.Count;
            while (CardsGiven.Count > 0)
                cards.Add(CardsGiven.Deal());
        }
    }
    if (totalCardsGiven == 0) {
        textBoxOnForm.Text += Name +
              " must draw from the stock." + Environment.NewLine;
        cards.Add(stock.Deal());
    }
}
```

这个AskForACard()方法检查每一个玩家（除了要牌的那个玩家），调用其DoYouHaveAny()方法，并把交来的牌增加到这手牌中。

如果没有交来的牌，玩家必须使用其Deal()方法从没发完的牌中取牌。

# 还有更多集合类型……

List和Dictionary对象是两个内置的泛型集合，它们是.NET框架的一部分。列表和字典非常灵活，可以按任何顺序访问其中的任何数据。不过，有时需要对程序处理数据的方式加以限制，因为真实世界中往往存在一些限制，通过程序表示时要与之类似。对于这种情况，要使用队列（Queue）或栈（Stack）。它们是与列表类似的另外两种泛型集合，不过它们特别擅长按某种特定的顺序处理数据。

还有其他类型的集合，不过栈和队列是最有可能用到的。

**如果所存储的第一个对象正是要使用的第一个对象，就要使用Queue，例如：**

- ★ 沿着单向街道行进的车流。
- ★ 排队的人。
- ★ 不挂机等候客户支持热线服务的客户。
- ★ 所有按先来先服务原则处理的情况。

队列是先入先出型（first-in first-out），这说明入队列的第一个对象也就是最先取出使用的对象。

**如果总想使用最近存储的对象，就要使用Stack，例如：**

- ★ 放在卡车后厢的家具。
- ★ 一摞书，你想先看最近增加的书。
- ★ 登机或下飞机的人。
- ★ 呈金字塔型的啦啦队，最顶上的人必须先撤下…… 想想看如果最底下的人先走开会多混乱！

栈是先进后出型（first-in last-out），最先进入栈的对象将最后一个取出。

## 泛型集合是.NET框架的一个重要部分

泛型集合确实有有用，正是因为它们如此重要，所以IDE会在增加到工程的每个类最上面都自动增加以下语句：

```
using System.Collections.Generic;
```

几乎所有大工程都会包含某种泛型集合，因为你的程序需要存储数据。处理真实世界中一组类似的事物时，大多可以很自然地归为一类，而这都能很好地对应为某种集合。

**队列类似于一个列表，可以在列表尾增加对象，而使用列表头的对象。栈只允许访问最后放入的对象。**

不过，可以使用foreach循环处理栈或队列，因为它们都实现了IEnumerable！

# 队列是FIFO——先进先出

队列很像列表，只不过不能任意地根据任何索引增加或删除项。向队列增加一个对象时，需要将其入队（enqueue）。这会把这个对象增加到队尾。可以从队头将第一个对象出队（dequeue）。此时，这个对象会从队列删除，队列中余下的对象会上移一个位置。

创建一个新的string队列。

```
Queue<string> myQueue = new Queue<string>();
myQueue.Enqueue("first in line");
myQueue.Enqueue("second in line");
myQueue.Enqueue("third in line");
myQueue.Enqueue("last in line");
string takeALook = myQueue.Peek(); ①
string getFirst = myQueue.Dequeue(); ②
string getNext = myQueue.Dequeue(); ③
int howMany = myQueue.Count; ④
myQueue.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Dequeue() returned: " + getFirst + "\n"
    + "The second Dequeue() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myQueue.Count);
                                             ⑤
```

在这里向队列增加4项。从队列中将它们取出时，还会按入队时的顺序出队。

Peek()允许"查看"队列中的第一项，但不把它删除。

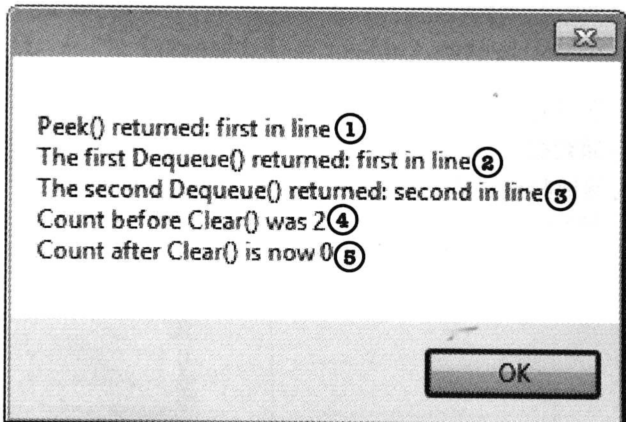
第一个Dequeue()会从队列中取出第一项。然后第二项会上移到第一的位置，再调用Dequeue()就会把这个对象取出。

Clear()方法清空队列中的所有对象。

队列的Count属性返回队列中的项数。



队列中的对象需要等待直到轮到它。队列中的第一个对象就是第一个出队的对象。



# 栈是LIFO——后进先出

栈与队列非常相似，但有一个显著区别。各个项要压入（push）栈，要从栈中取出一项时，需要从栈中弹出(pop)。从栈中弹出一项时，会得到最近压入的那一项。这有些像一摞盘子、杂志之类的东西，可以向栈顶放东西，但是需要把顶上的东西拿开才能拿到它下面的东西。

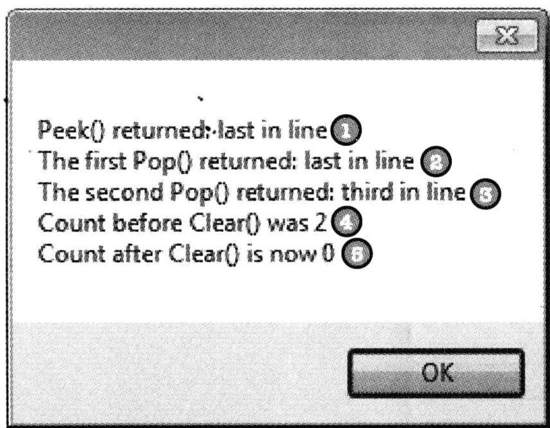
栈的创建类似于创建所有其他泛型集合。

将一项压入栈时，它会把其他项向下压一个位置，而留在栈顶。

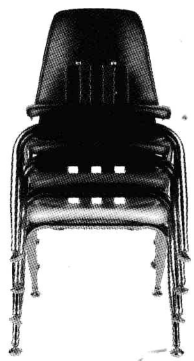
```
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");
(1)string takeALook = myStack.Peek();
(2)string getFirst = myStack.Pop();
(3)string getNext = myStack.Pop();
(4)int howMany = myStack.Count;
myStack.Clear();
MessageBox.Show("Peek() returned: " + takeALook + "\n"
    + "The first Pop() returned: " + getFirst + "\n"
    + "The second Pop() returned: " + getNext + "\n"
    + "Count before Clear() was " + howMany + "\n"
    + "Count after Clear() is now " + myStack.Count);
                                                 (5)
```

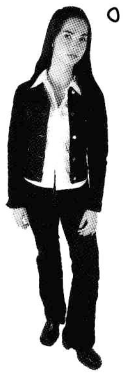
从栈中弹出一项时，会得到最近增加的那一项。

这里还可以使用Environment.NewLine而不是\n，不过我们希望代码能更易于阅读。




最后放入栈的元素是第一个取出的对象。





等一下，我有一个问题。你还没有告诉我哪些事情用栈或队列能做到而用列表做不到。对我来讲，它们只是能让我少写几行代码。但我无法得到位于栈或队列中间的项。而使用列表可以很轻松地做到这一点！那又何必为了这么一点点方便去使用栈或队列呢？

**别担心，使用队列或栈时不会放弃任何功能。**

将一个Queue对象转换为List对象相当容易。而且把List转换为Queue，或者把Queue转换为Stack……也同样很简单。实际上，任何实现了**IEnumerable**接口的对象都可以创建List、Queue或Stack。只需使用重载构造函数，并把希望复制的集合作为参数传入。这说明你可以灵活性和方便性兼顾，用最适合的集合来表示数据。（不过要记住，你建立的是一个副本，这说明你会创建一个全新的对象，并把它增加到堆中）。

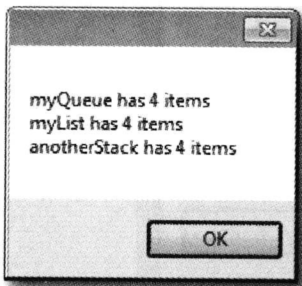
下面建立一个包含4项的栈，在这里，这是一个string栈。

```
Stack<string> myStack = new Stack<string>();
myStack.Push("first in line");
myStack.Push("second in line");
myStack.Push("third in line");
myStack.Push("last in line");

Queue<string> myQueue = new Queue<string>(myStack);
List<string> myList = new List<string>(myQueue);
Stack<string> anotherStack = new Stack<string>(myList);
MessageBox.Show("myQueue has " + myQueue.Count + " items\n"
    + "myList has " + myList.Count + " items\n"
    + "anotherStack has " + anotherStack.Count + " items\n");
```

可以很容易地把这个栈转换为一个队列，然后把队列复制到一个列表，再把列表复制到另一个栈。

所有4项都复制到新集合中。




**……而且总能用一个foreach循环访问栈或队列中的所有成员！**

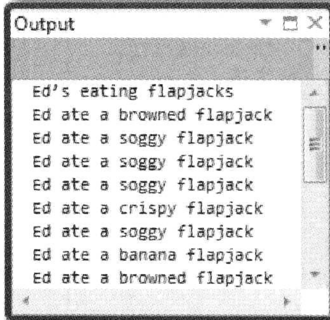



编写一个程序来帮助一个餐厅，餐厅里坐满了吃煎饼的伐木工人。先来完成Lumberjack类，填入缺少的代码。然后设计窗体，并增加按钮的事件处理程序。

❶ 以下是**Lumberjack**类。填写FlapjackCount的获取存取方法以及TakeFlapjacks和EatFlapjacks方法。

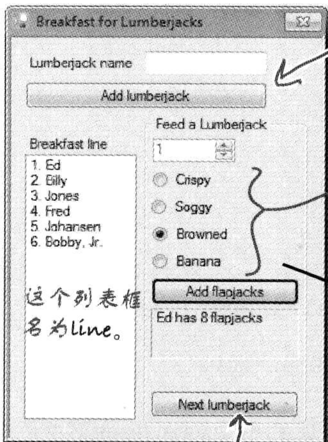
```
class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;
    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }
    public int FlapjackCount { get { // return the count } }
    public void TakeFlapjacks(Flapjack Food, int HowMany) {
       // Add some number of flapjacks to the Meal stack
    }
    public void EatFlapjacks() {
       // Write this output to the console
    }
}
```

```
enum Flapjack {
    Crispy,
    Soggy,
    Browned,
    Banana
}
```




❷ 创建这个窗体。允许在一个文本框中输入伐木工人的名字，让他们吃早餐时排队。可以向排头的伐木工人发一个煎饼盘子。然后使用“Next lumberjack”按钮告诉他前移，去吃这些煎饼。我们已经提供了“Add flapjacks”按钮的点击事件处理程序。要使用一个名为**breakfastLine**的队列跟踪这些伐木工人。

**注意Flapjack enum使用了大写字母(“Soggy”)，但输出却是小写字母(“soggy”)，这是怎么回事？下面这个提示可以帮助你得到正确的输出。ToString()返回一个string对象，它的一个公共成员ToLower()方法可以返回字符串的一个小写版本。**




用户点击“Add Lumberjack”时，将name文本框中的人名增加到BreakfastLine队列。

把这些RadioButton控件拖到groupbox时，窗体会自动把它们链接起来，只允许用户一次选择其中一个。查看addFlapjacks_Click方法来了解应当如何对它们命名。

这个列表框名为line。

这个按钮将下一个伐木工人出队，调用其EatFlapjacks方法，然后重绘这个列表框。

```
private void addFlapjacks_Click(...) {
    Flapjack food;
    if (crispy.Checked == true)
        food = Flapjack.Crispy;
    else if (soggy.Checked == true)
        food = Flapjack.Soggy;
    else if (browned.Checked == true)
        food = Flapjack.Browned;
    else
        food = Flapjack.Banana;

    Lumberjack currentLumberjack = breakfastLine.Peek();
    currentLumberjack.TakeFlapjacks(food,
                     (int)howMany.Value);
    RedrawList();
}
```

注意这个特殊的“else if”语法。

Peek()返回队列中第一个伐木工人的引用。

NumericUpDown控件名为howMany，标签名为nextInLine。

需要增加一个RedrawList()方法用队列的内容更新列表框。3个按钮都要调用这个方法。这里给出一个提示：它使用了一个foreach循环。




```
private Queue<Lumberjack> breakfastLine = new Queue<Lumberjack>();
private void addLumberjack_Click(object sender, EventArgs e) {
    breakfastLine.Enqueue(new Lumberjack(name.Text));
    name.Text = "";
    RedrawList();
}
private void RedrawList() {
    int number = 1;
    line.Items.Clear();
    foreach (Lumberjack lumberjack in breakfastLine) {
        line.Items.Add(number + ". " + lumberjack.Name);
        number++;
    }
    if (breakfastLine.Count == 0) {
        groupBox1.Enabled = false;
        nextInLine.Text = "";
    } else {
        groupBox1.Enabled = true;
        Lumberjack currentLumberjack = breakfastLine.Peek();
        nextInLine.Text = currentLumberjack.Name + " has "
            + currentLumberjack.FlapjackCount + " flapjacks";
    }
}
private void nextLumberjack_Click(object sender, EventArgs e) {
    Lumberjack nextLumberjack = breakfastLine.Dequeue();
    nextLumberjack.EatFlapjacks();
    nextInLine.Text = "";
    RedrawList();
}

class Lumberjack {
    private string name;
    public string Name { get { return name; } }
    private Stack<Flapjack> meal;

    public Lumberjack(string name) {
        this.name = name;
        meal = new Stack<Flapjack>();
    }

    public int FlapjackCount { get { return meal.Count; } }

    public void TakeFlapjacks(Flapjack food, int howMany) {
        for (int i = 0; i < howMany; i++) {
            meal.Push(food);
        }
    }

    public void EatFlapjacks() {
        Console.WriteLine(name + "'s eating flapjacks");
        while (meal.Count > 0) {
            Console.WriteLine(name + " ate a "
                + meal.Pop().ToString().ToLower() + " flapjack");
        }
    }
}
```

列表框名为"line"，两个按钮之间的标签名为"nextInLine"。

RedrawList()方法使用一个foreach循环从队列取出伐木工人，并将它们分别增加到列表框。

这个if语句用队列中第一个伐木工人的有关信息来更新标签。

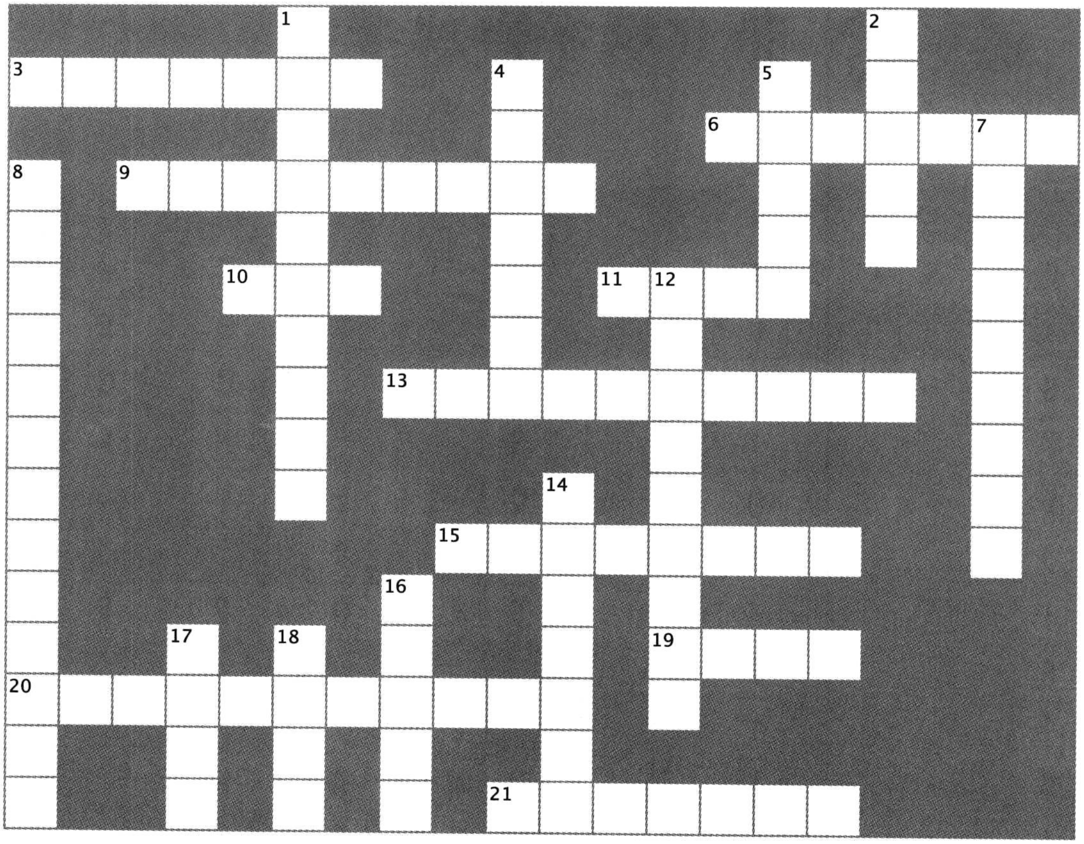
TakeFlapjacks方法更新Meal栈。

在这里将Flapjack enum转换为小写。花点时间来了解到底做了什么。

EatFlapjacks方法使用一个while循环打印出伐木工人的用餐情况。

meal.Pop()返回一个enum，然后调用它的ToString()方法返回一个string对象，再调用这个string对象的ToLower()方法返回另一个string对象。

# 集合填字游戏



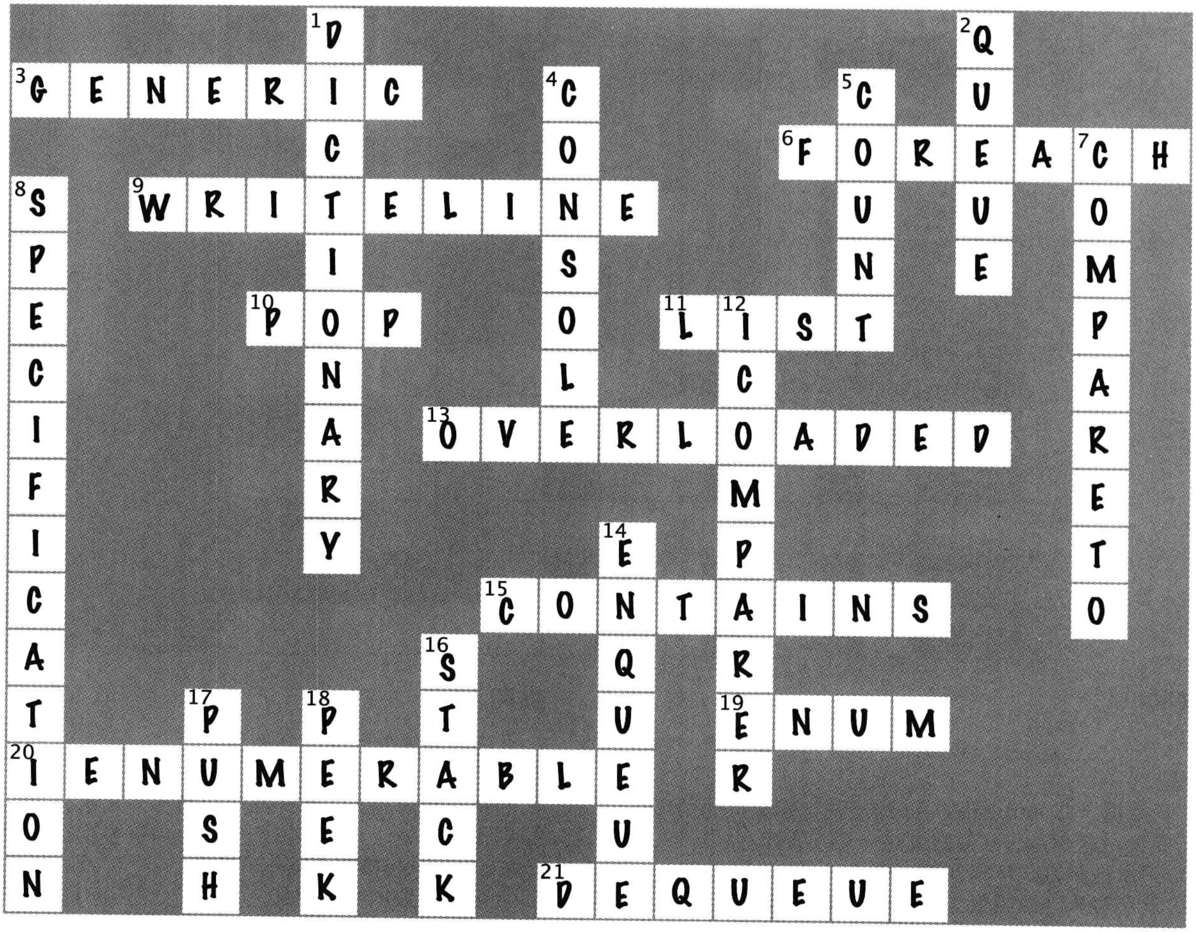
## 横向

3. ________ 集合的一个实例只能处理一种特定的类型。
6. 只能用于 IEnumerable<T>的一种特殊类型的循环。
9. 要使用这个方法向输出发送一个字符串。
10. 采用这种方式从栈删除一项。
11. 这个对象与数组很类似，但是更为灵活。
13. 类中同名但有不同参数的两个方法是________。
15. 确定某个对象是否在一个集合的方法。
19. 一种记录不同类别的简便方法。
20. 所有泛型集合都实现了这个接口。
21. 采用这种方式从队列删除一项。

## 纵向

1. 允许将键映射到值的泛型集合。
2. 这个集合是先进先出的。
4. 利用这个内置类，允许程序向输出写文本。
5. 利用这个方法可以得出一个集合中有多少个元素。
7. IComparable接口中唯一的方法。
8. 大多数专业的工程都从________开始。
12. 实现了这个接口的对象可以帮助你对列表内容排序。
14. 采用这种方式向队列增加一项。
16. 这个集合是先进后出的。
17. 采用这种方式向栈增加一项。
18. 这个方法返回栈或队列的下一个对象。

# 集合填字游戏答案


1 DICTIONARY
2 QUEUE
3 GENERIC
4 CONSOLE
5 COUNT
6 FOREACH
7 COMPARETO
8 SPECIFICATION
9 WRITELINE
10 POP
11 LIST
12 ICOMPARER
13 OVERLOADED
14 ENQUEUE
15 CONTAINS
16 STACK
17 PUSH
18 PEEK
19 ENUM
20 IENUMERABLE
21 DEQUEUE

姓名：　日期：

# C#实验室

## 冒险游戏

这个实验室将给出一个程序的规范，要求你根据前几章所学的知识构建这个程序。

这个工程比你之前见过的所有工程规模更大。所以在着手开发之前，先花点时间了解整个问题，磨刀不误砍柴工。另外，如果你被某个问题卡住了也不用担心，这里并没有新内容，所以完全可以继续向下读，以后再回来完成这个实验室。

我们已经为你完成了一些设计细节，而且可以保证已经提供了你需要的全部知识……再没有其他可以补充的了。

**你要自己来完成这个任务。**可以从网站下载这个实验室的一个可执行文件…… 不过我们不会提供这个工程的源代码。

# 规范：构建一个冒险游戏

你的任务是构建一个冒险游戏，一个勇猛的冒险家在探险，一关一关地打败一些危险的敌人。你要构建一个按回合进行的系统，也就是说，玩家走一步，然后敌人走一步。玩家可以移动或攻击，接下来各个敌人得到机会移动并攻击。游戏会一直进行下去，直到玩家打败7关的所有敌人，或者被敌人打死。

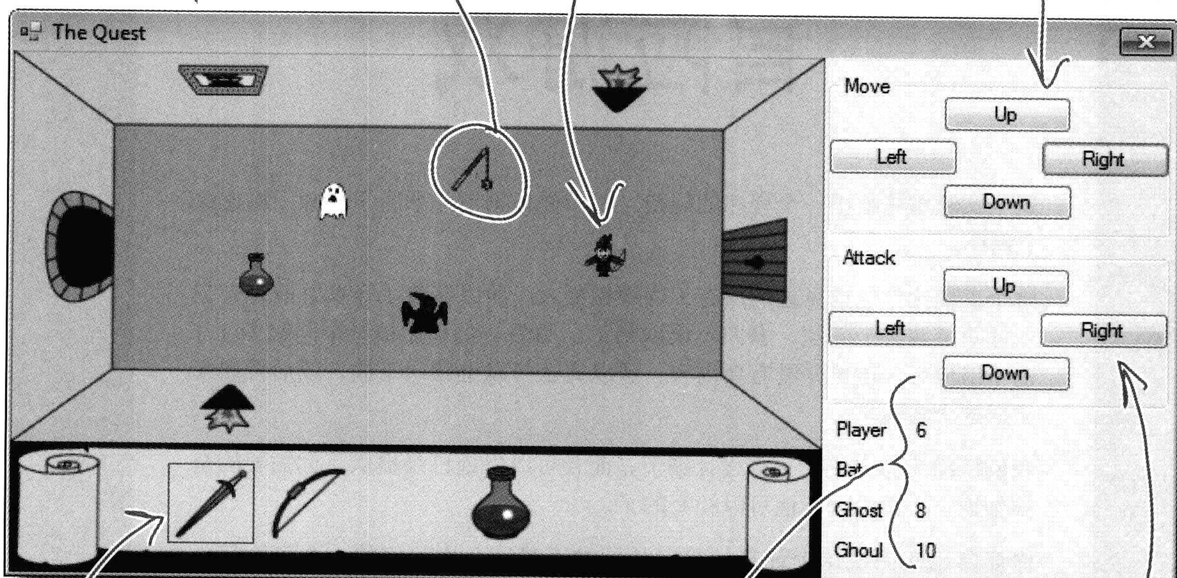
敌人有一点优势，他们每个回合都能移动，而且移动后如果在攻击范围内就会攻击玩家。

这个游戏窗口给出了地牢的俯视图，玩家在这里与敌人展开战斗。

玩家一路上可以选择武器和补药。

玩家和敌人在地牢里来回移动。

玩家使用4个Move按钮移动。



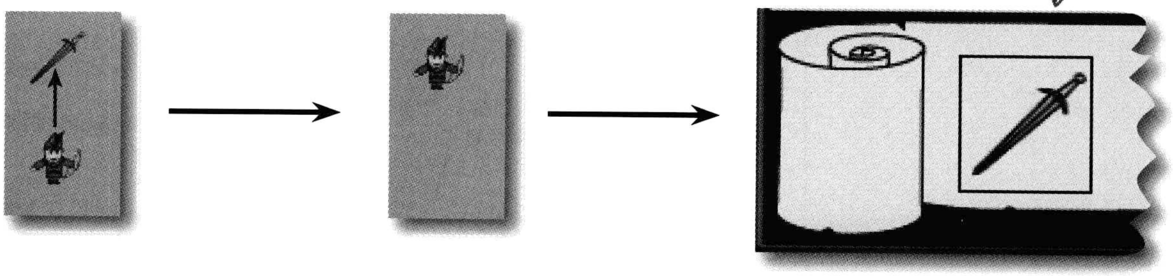

这是玩家的武器清单，显示了玩家挑选的武器和补药，另外他目前使用的武器（补药）周围画有一个方框。玩家点击其中一项作为自己的装备，然后使用Attack按钮使用这个装备。

这个游戏会显示玩家和敌人的点数。如果玩家攻击了敌人，敌人的点数就会下降。一旦敌人（或玩家）的点数下降到0，他就会死掉。

这4个按钮用来攻击敌人以及喝补药（玩家可以使用任意按钮喝补药。

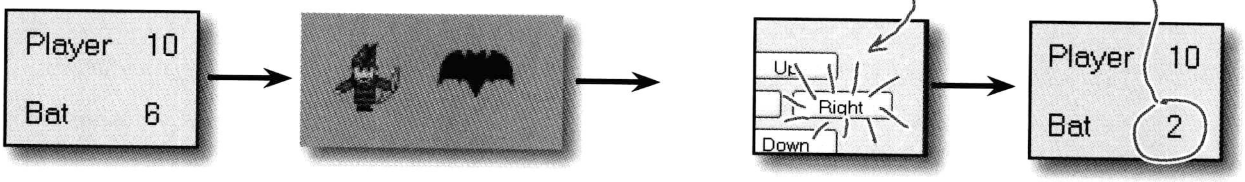
## 玩家挑选武器……

地牢的四周散布着一些武器和补药，玩家(Player)可以挑选这些装备用来打败他的敌人。他要做的就是移动到一个武器，武器就会从地面上消失而出现在他的武器清单里。



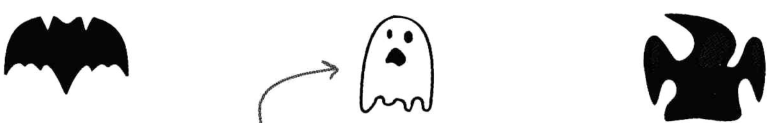

## ……并用这些武器攻击敌人

游戏中，每一关都会提供一个武器供玩家选择，用来打败他的敌人。一旦挑选了武器，武器就要从地面上消失。



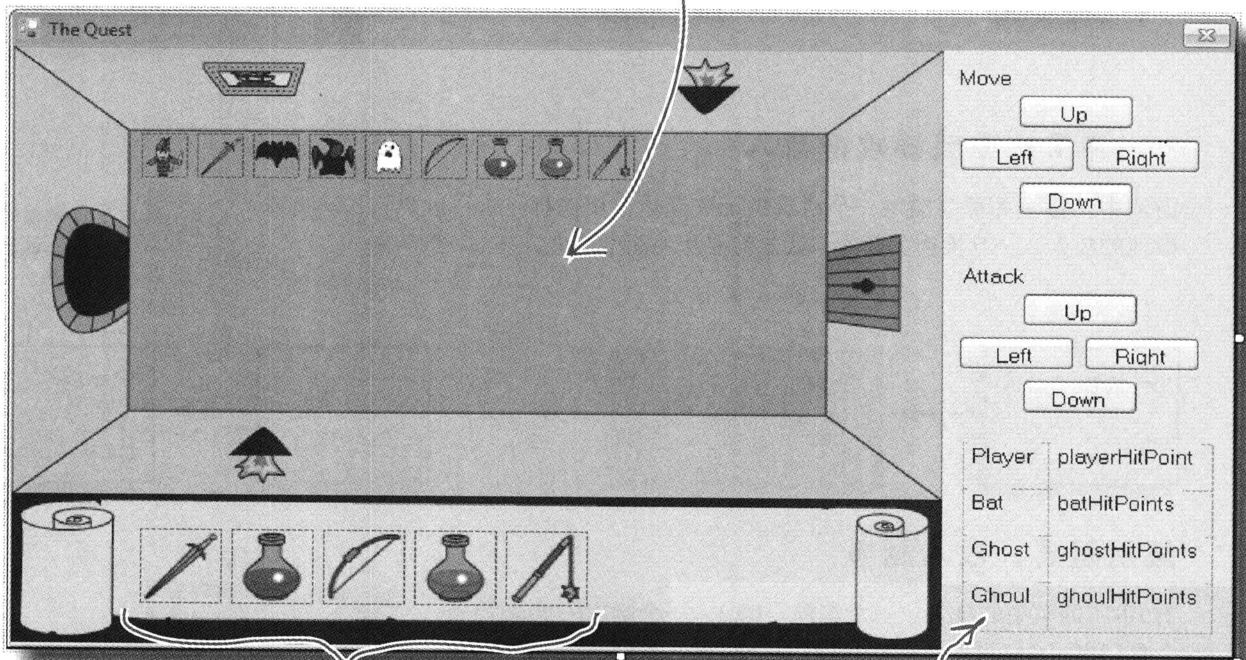

## 级别越高，敌人越多

有3种不同类型的敌人：一个蝙蝠（Bat)、一个幽灵(Ghost)和一个食尸鬼（Ghoul）。第1关只有一个蝙蝠，第7关是最后一关，这3个敌人都会出现。



## 设计：创建窗体

这个游戏看起来很独特，它的外观要通过窗体来提供。使用窗体的BackgroundImage属性显示地牢和武器清单的图像，另外使用一系列PictureBox控件在地牢中显示玩家、武器和敌人。要用一个TableLayoutPanel控件显示玩家、蝙蝠、幽灵和食尸鬼的点数，另外用来完成移动和攻击的按钮也放在这个TableLayoutPanel控件中。

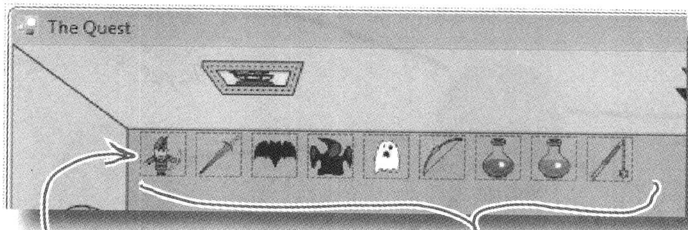



**可以从Head First Labs网站（www.headfirstlabs.com/books/hfcsharp）下载背景图像以及武器、敌人和玩家的图片。**

## 地牢里所有东西都是PictureBox

玩家、武器和敌人都应当用图标表示。增加9个`PictureBox`控件，设置其`Visible`属性为`False`。游戏中可以移动这些控件，并根据需要切换它们的`Visible`属性。

可以设置PictureBox的BackColor属性为Color.Transparent，使窗体的背景图片或颜色能透过这个图片中透明的像素显示出来。




向地牢增加9个PictureBox 控件。使用Size属性使每个控件大小为30×30。放在哪里并不重要，窗体会移动这些控件。点击PictureBox会显示一个小黑箭头，使用这个小黑箭头为各个控件设置从Head First Labs网站下载的某个图片。

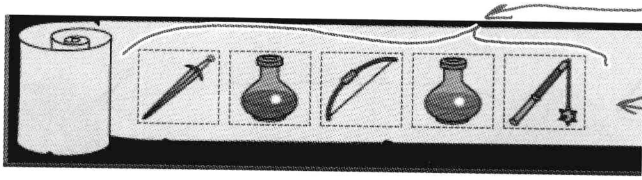
增加了9个PictureBox控件后，右键点击玩家的图标，选择“Bring to Front”，将3个武器图标置于底层。这样可以确保玩家图标总在他选择的装备“上面”。

IDE中控件会相互重叠，所以窗体需要知道哪些控件在上层，哪些在下层。“Bring to Front”和“Send to Back”窗体设计命令就用作这个目的。

## 武器清单也包含PictureBox控件

可以将玩家的武器清单用5个50×50的`PictureBox`控件表示。将各个控件的`BackColor`属性设置为**`Color.Transparent`**（如果使用属性窗口来设置这个属性，只需在`BackColor`行中键入`Color.Transparent`）。由于图片文件背景透明，所以可以看到图片下层的卷轴和地牢。

还需要5个50x50的PictureBoxes表示武器清单。

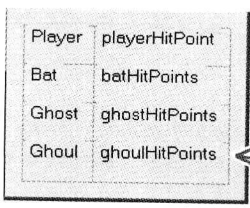


玩家装备了某个武器时，窗体应将这个武器图标的BorderStyle属性设置为FixedSingle，而其余武器图标的BorderStyle设置为None。

## 创建统计窗口

点数在一个`TableLayoutPanel`中显示，攻击和移动按钮也放在这个面板中。为显示点数，在面板中创建两列，将分栏条向左拖一点点。增加4行，每行高度为25%，并向8个单元格分别增加一个`Label`控件。

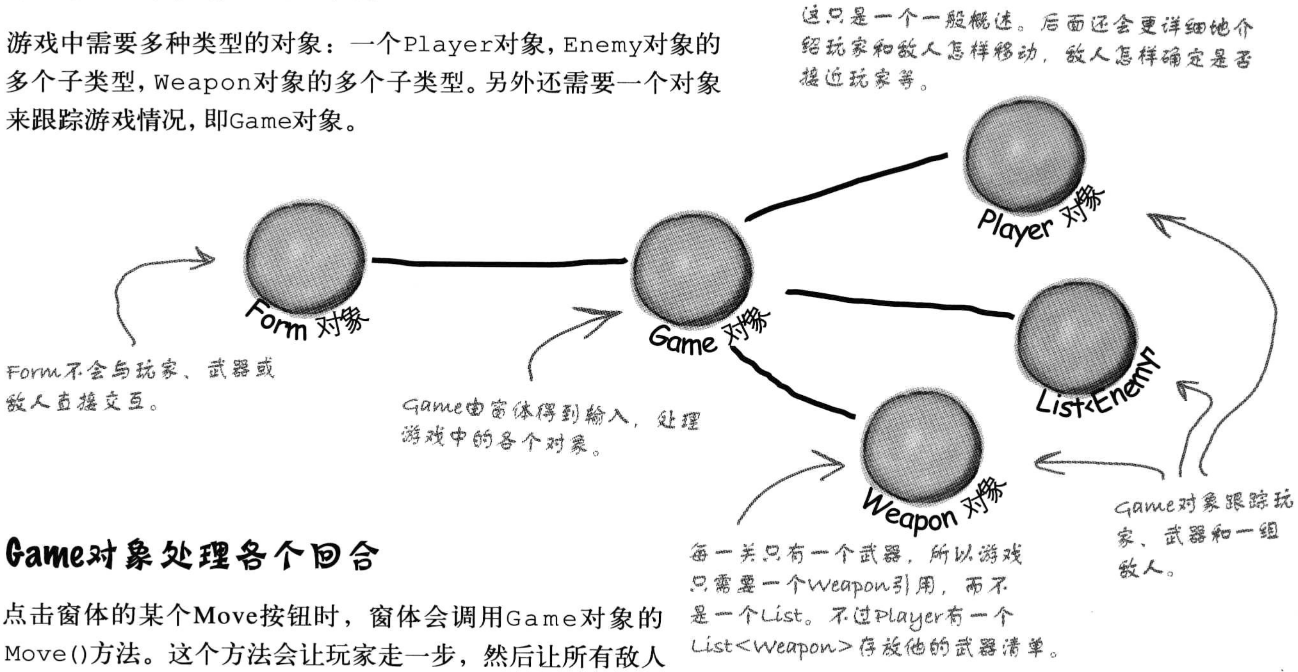
2列，4行……用8个单元格表示点数统计结果。




每个单元格中有一个Label，游戏中可以更新这些值。

# 体系结构：使用对象

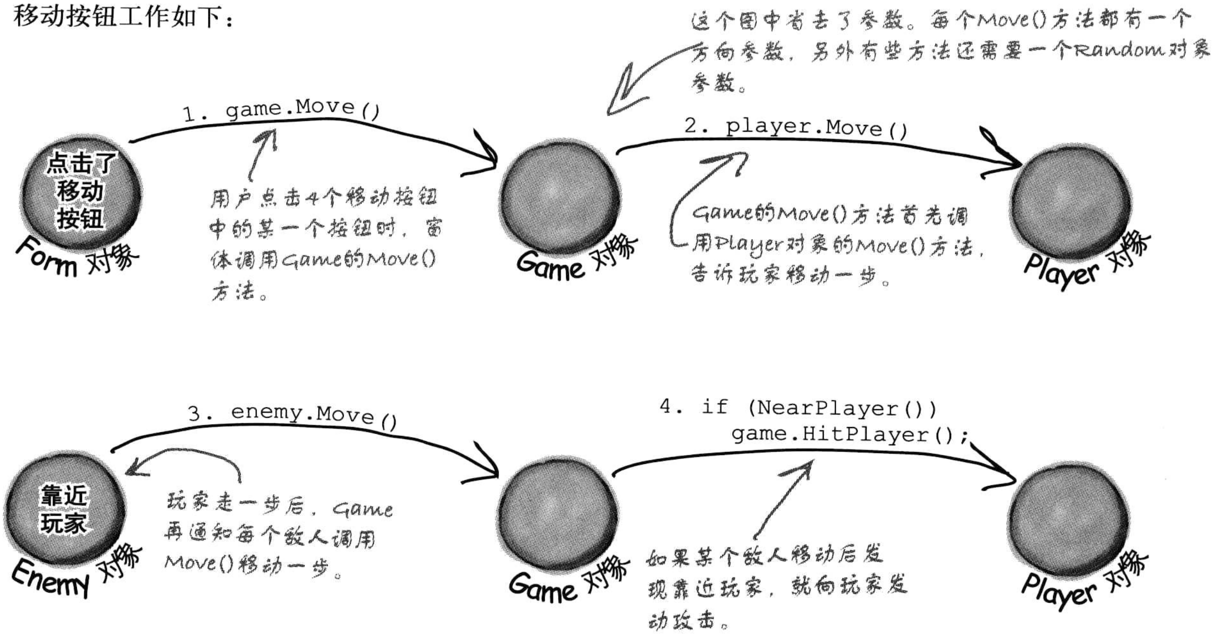
游戏中需要多种类型的对象：一个Player对象，Enemy对象的多个子类型，Weapon对象的多个子类型。另外还需要一个对象来跟踪游戏情况，即Game对象。




## Game对象处理各个回合

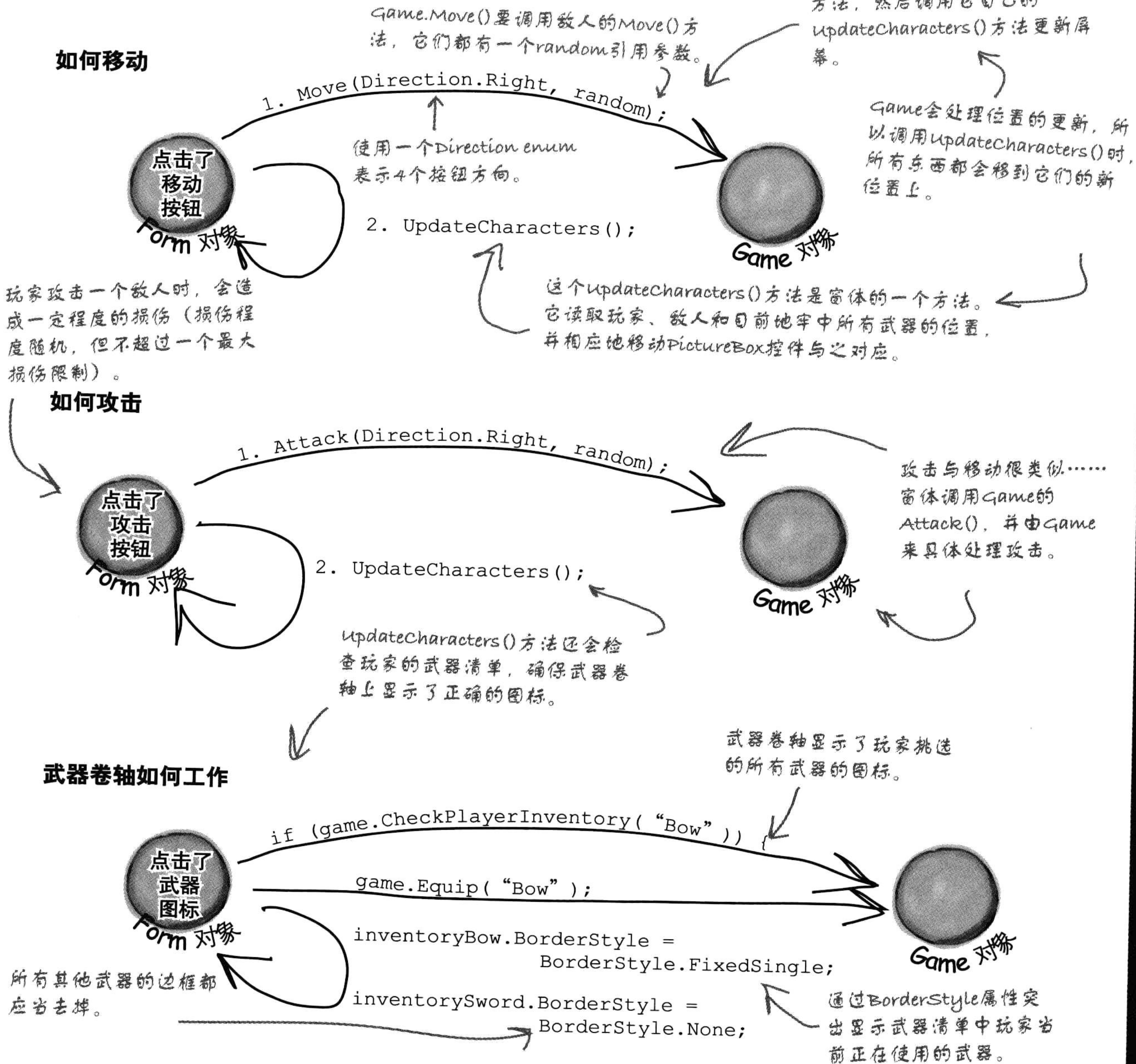
点击窗体的某个Move按钮时，窗体会调用Game对象的Move()方法。这个方法会让玩家走一步，然后让所有敌人走一步。所以要由Game来处理游戏中每一回合的移动。

例如，移动按钮工作如下：



# 窗体把活动委托给Game对象

移动、攻击和武器清单的工作都从窗体开始。所以如果点击一个移动或攻击按钮，或者点击武器清单中的一个武器时，就会触发窗体中的相应代码。但是真正控制游戏中各个对象的却是Game对象。所以窗体必须把发生的所有一切都告诉Game对象，然后Game对象接手它的工作：



# 构建Game类

编写Game类时可以把以下代码作为起点。还有很多工作需要你来完成，所以仔细查看这个代码，把它们键入到IDE中，做好准备开始吧。

```
using System.Drawing;

class Game {
  public List<Enemy> Enemies;
  public Weapon WeaponInRoom;

  private Player player;
  public Point PlayerLocation { get { return player.Location; } }
  public int PlayerHitPoints { get { return player.HitPoints; } }
  public List<string> PlayerWeapons { get { return player.Weapons; } }

  private int level = 0;
  public int Level { get { return level; } }

  private Rectangle boundaries;
  public Rectangle Boundaries { get { return boundaries; } }

  public Game(Rectangle boundaries) {
    this.boundaries = boundaries;
    player = new Player(this,
      new Point(boundaries.Left + 10, boundaries.Top + 70));
  }
  public void Move(Direction direction, Random random) {
    player.Move(direction);
    foreach (Enemy enemy in Enemies)
      enemy.Move(random);
  }
  public void Equip(string weaponName) {
    player.Equip(weaponName);
  }
  public bool CheckPlayerInventory(string weaponName) {
    return player.Weapons.Contains(weaponName);
  }
  public void HitPlayer(int maxDamage, Random random) {
    player.Hit(maxDamage, random);
  }
```

需要System.Drawing中的Rectangle和Point类，所以一定要在类的最前面增加这行代码。

如果Enemy和Weapon得到了很好的封装，这些作为公共属性是可以的……换句话说，要确保窗体不能对它们做不适当的操作。

游戏维护着一个私有的Player对象。窗体只能通过Game的方法与这个对象交互，而不能直接交互。

Rectangle对象有Top, Bottom, Left和Right字段，非常适于表示整个游戏区。

Game首先有一个边框表示地牢，另外在地牢中创建一个新的Player对象。

移动很简单：让玩家按窗体提供的方向移动，再让各个敌人按随机的方向移动。

这些都是很好的封装例子……Game不知道Player如何处理这些动作，它只是传递所需的信息，让Player来完成其余的工作。

```
    public void IncreasePlayerHealth(int health, Random random) {
      player.IncreaseHealth(health, random);
    }

    public void Attack(Direction direction, Random random) {
      player.Attack(direction, random);
      foreach (Enemy enemy in Enemies)
        enemy.Move(random);
    }

    private Point GetRandomLocation(Random random) {
      return new Point(boundaries.Left +
        random.Next(boundaries.Right / 10 - boundaries.Left / 10) * 10,
            boundaries.Top +
        random.Next(boundaries.Bottom / 10 - boundaries.Top / 10) * 10);
    }

    public void NewLevel(Random random) {
      level++;
      switch (level) {
        case 1:
          Enemies = new List<Enemy>();
          Enemies.Add(new Bat(this, GetRandomLocation(random)));
          WeaponInRoom = new Sword(this, GetRandomLocation(random));
          break;
      }
    }
  }
```

Attack()与Move()几乎相同。玩家发动攻击，接下来轮到敌人移动。

GetRandomLocation()将在NewLevel()方法中用到，NewLevel()方法使用GetRandomLocation()来确定把敌人和武器放在哪里。

这只是一个数学技巧，在表示地牢区的矩形内得到一个随机位置。

我们只增加了第1关的相应case语句。要由你来增加其他几关的case语句。

我们在武器清单中只留出了一瓶蓝色补药和一瓶红色补药的空间。所以如果玩家已经有一瓶红色补药，游戏就不能再在这一关增加红色补药（蓝色补药也是如此）。

## 完成其他关的代码

你要完成这个NewLevel()方法。下面是对各关的一个概要说明：

| 第几关 | 敌人 | 武器 |
|---|---|---|
| 2 | 幽灵 | 蓝色补药 |
| 3 | 食尸鬼 | 弓箭 |
| 4 | 蝙蝠，幽灵 | 如果第3关没有选择弓箭，则是弓箭，否则为蓝色补药 |
| 5 | 蝙蝠，食尸鬼 | 红色补药 |
| 6 | 幽灵，食尸鬼 | 重头棒 |
| 7 | 蝙蝠，幽灵，食尸鬼 | 如果第6关没有选择重头棒，则是重头棒，否则为红色补药 |
| 8 | N/A | N/A - 使用Application.Exit()结束游戏 |

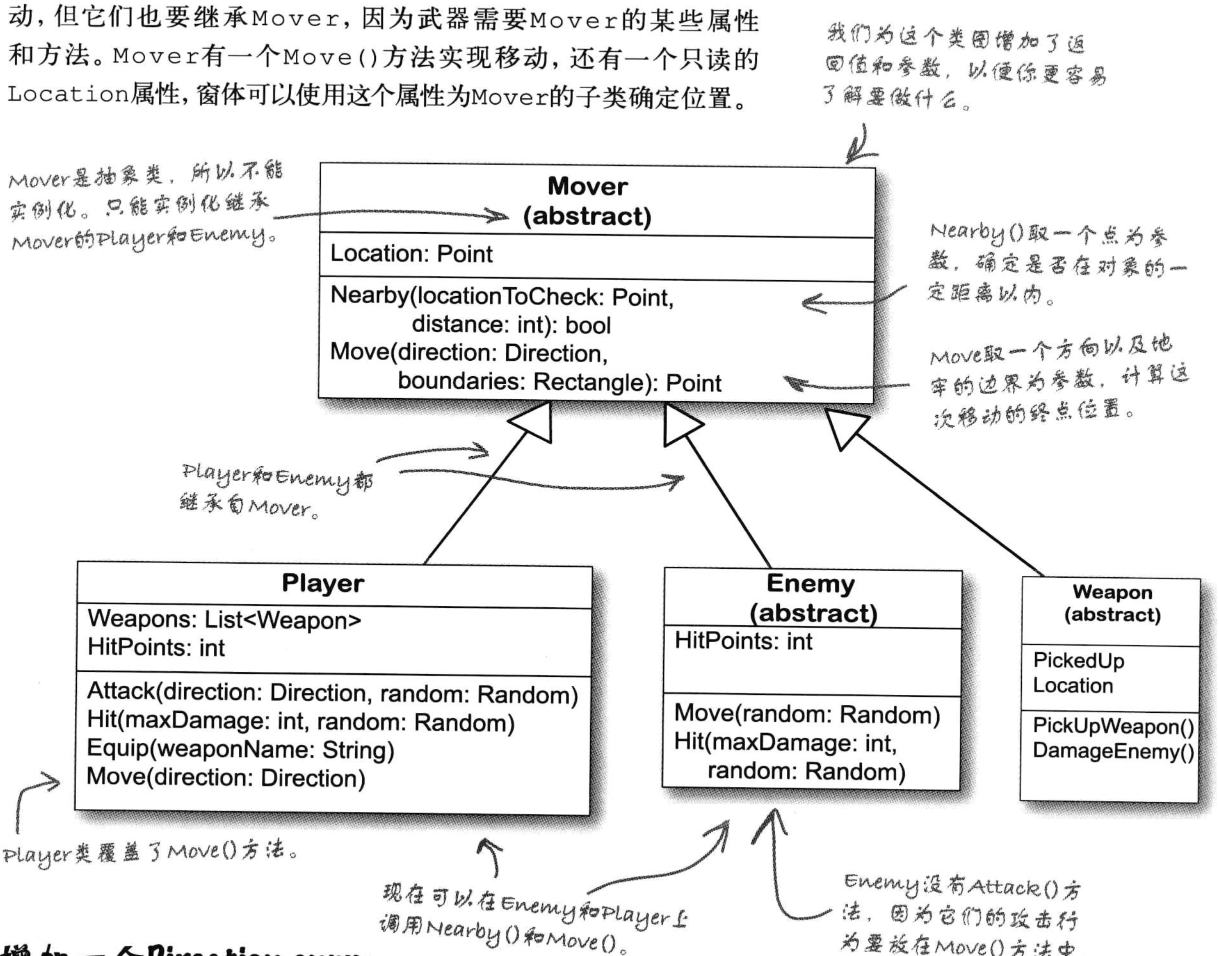
所以如果玩家在第2关得到了一瓶蓝色补药，而且仍在他的武器清单里，那么这一关什么也不会出现。

只有当第5关得到的红色补药已经用完时这才会显示。

## 寻找共同行为：移动

你已经知道了，重复代码是不好的，而且如果两个或多个对象有相同的行为时总会出现重复代码。在这个地牢游戏中也是如此…… 敌人和玩家都要移动。

下面来创建一个Mover类，把敌人和玩家共同的行为抽取到一处。Player和Enemy都继承自Mover。另外，尽管武器不能移动，但它们也要继承Mover，因为武器需要Mover的某些属性和方法。Mover有一个Move()方法实现移动，还有一个只读的Location属性，窗体可以使用这个属性为Mover的子类确定位置。




## 增加一个Direction enum

Mover类以及其他的许多类都需要一个Direction enum。创建这个enum，提供4个枚举值：Up、Down、Left和Right。

# Mover类源代码

以下是Mover类的源代码:

```
abstract class Mover {
  private const int MoveInterval = 10;
  protected Point location;
  public Point Location { get { return location; } }
  protected Game game;

  public Mover(Game game, Point location) {
    this.game = game;
    this.location = location;
  }

  public bool Nearby(Point locationToCheck, int distance) {
    if (Math.Abs(location.X - locationToCheck.X) < distance &&
       (Math.Abs(location.Y - locationToCheck.Y) < distance)) {
      return true;
    } else {
      return false;
    }
  }
  public Point Move(Direction direction, Rectangle boundaries) {
    Point newLocation = location;
    switch (direction) {
      case Direction.Up:
        if (newLocation.Y - MoveInterval >= boundaries.Top)
          newLocation.Y -= MoveInterval;
        break;
      case Direction.Down:
        if (newLocation.Y + MoveInterval <= boundaries.Bottom)
          newLocation.Y += MoveInterval;
        break;
      case Direction.Left:
        if (newLocation.X - MoveInterval >= boundaries.Left)
          newLocation.X -= MoveInterval;
        break;
      case Direction.Right:
        if (newLocation.X + MoveInterval <= boundaries.Right)
          newLocation.X += MoveInterval;
        break;
      default: break;
    }
    return newLocation;
  }
}
```

由于protected属性只能由子类访问，Form对象不能设置location……，只能通过我们定义的公共获取存取方法来读取这个属性。

Mover的实例取Game对象和一个当前位置作为参数。

Nearby方法根据这个对象的当前位置检查一个Point。如果它们之间的距离不超过**distance**，则返回true，否则返回false。

Move()方法尝试向某个方向移动一步。如果可以移动，则返回新的Point。如果达到边界，则返回原来的Point。

如果最终位置在边界以外，新位置仍为原来的起始点。

最后，返回这个新位置（有可能还是原来的起始位置）！

## Player类跟踪玩家

下面来看Player类。在IDE中键入下面的代码，以此为起点，准备补充。

Player和Enemy对象都要保持在地牢以内，这说明它们需要知道游戏区的边界。使用边界Rectangle的Contains()方法来确保它们没有超出边界。

```
class Player : Mover {
  private Weapon equippedWeapon;
  private int hitPoints;
  public int HitPoints { get { return hitPoints; } }

  private List<Weapon> inventory = new List<Weapon>();
  public List<string> Weapons {
    get {
      List<string> names = new List<string>();
      foreach (Weapon weapon in inventory)
        names.Add(weapon.Name);
      return names;
    }
  }

  public Player(Game game, Point location);
        : base(game, location) {
    hitPoints = 10;
  }

  public void Hit(int maxDamage, Random random) {
    hitPoints -= random.Next(1, maxDamage);
  }

  public void IncreaseHealth(int health, Random random) {
    hitPoints += random.Next(1, health);
  }

  public void Equip(string weaponName) {
    foreach (Weapon weapon in inventory) {
      if (weapon.Name == weaponName)
        equippedWeapon = weapon;
    }
  }
}
```

Player的所有属性都是隐藏的，不允许直接访问。

Player在武器清单中可以保存多个武器，不过一次只能装备其中的一个武器。

Player继承自Mover，所以这会把Game和位置传入基类。

玩家的构造函数将hitPoints设置为10，然后调用基类构造函数。

敌人攻击玩家时，会造成一定程度的损伤（损伤程度随机）。服用补药时，玩家的健康度会增加（增加量随机）。

Equip()方法告诉玩家实际装备某个武器。点击武器清单里的某个图标时，Game对象会调用这个方法。

Player对象一次只能装备一个Weapon对象。

尽管补药只能帮助玩家而不是伤害敌人，但游戏仍认为它们是武器。这样一来，武器清单就可以是一个List<Weapon>，游戏可以用其WeaponInRoom引用指向其中一个武器。

# 为Player编写Move()

Game调用Player的Move()方法通知玩家朝某个方向移动。Move()取移动方向作为参数（使用你应该已经增加的Direction enum）。下面是这个方法的开始部分：

点击窗体上某个移动按钮时会调用Game对象的这个方法。

```
public void Move(Direction direction) {
  base.location = Move(direction, game.Boundaries);
  if (!game.WeaponInRoom.PickedUp) {
    // see if the weapon is nearby, and possibly pick it up
  }
}
```

Move是Mover基类的方法。

玩家选中一个武器时，这个武器要从地牢消失，出现在武器清单里。

你要填完这个方法的其余代码。查看武器是否靠近玩家（在某个距离以内）。如果是，则选择这个武器，把它增加到玩家的武器清单中。

玩家选择了一个武器时，Weapon和窗体会置武器的Picturebox不可见……这不是Player类的任务。

如果玩家只有这一样武器，就立即装备这个武器。这样一来，下一回合玩家就可以使用这个武器了。

## 再增加Attack()方法

武器都有一个Attack()方法，取一个Direction enum和一个Random对象作为参数。玩家的Attack()方法会确定目前装备了什么武器，并调用这个武器的Attack()方法。

接下来是Attack()方法。点击窗体的某个攻击按钮时，会调用这个方法，它需要一个方向参数（同样地，这是一个Direction enum值）。以下是这个方法的签名：

```
public void Attack(Direction direction, Random random) {
  // Your code goes here
}
```

如果武器是一瓶补药，玩家喝完补药后Attack()将把这瓶补药从武器清单中删除。

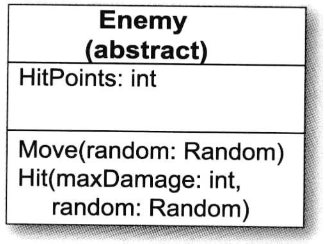
如果玩家没有装备任何武器，这个方法什么也不做。如果玩家确实装备了一个武器，就要调用这个武器的Attack()方法。

但是补药比较特殊。如果使用的是补药，要把它从玩家的武器清单中删除，因为补药已经喝完了。

Potion要实现一个IPotion接口（稍后将做更多说明），所以可以使用“is”来看一个Weapon是否实现了IPotion。

# Bat、Ghost和Ghoul继承自Enemy类

下面给出另一个有用的抽象类：Enemy。各种不同类型的敌人都有自己的类，这些类都继承自Enemy类。不同类型的敌人以不同的方式移动，所以Enemy抽象类中Move方法是一个抽象方法，3个Enemy子类需要根据它们如何移动以不同的方式实现这个方法。




```
abstract class Enemy : Mover {
  private const int NearPlayerDistance = 25;
  private int hitPoints;
  public int HitPoints { get { return hitPoints; } }
  public bool Dead { get {
      if (hitPoints <= 0) return true;
      else return false;
    }
  }
  public Enemy(Game game, Point location, int hitPoints)
      : base(game, location) { this.hitPoints = hitPoints; }

  public abstract void Move(Random random);

  public void Hit(int maxDamage, Random random) {
    hitPoints -= random.Next(1, maxDamage);
  }

  protected bool NearPlayer() {
    return (Nearby(game.PlayerLocation,
                   NearPlayerDistance));
  }
  protected Direction FindPlayerDirection(Point playerLocation) {
    Direction directionToMove;
      if (playerLocation.X > location.X + 10)
         directionToMove = Direction.Right;
    else if (playerLocation.X < location.X - 10)
         directionToMove = Direction.Left;
    else if (playerLocation.Y < location.Y - 10)
         directionToMove = Direction.Up;
    else
         directionToMove = Direction.Down;
    return directionToMove;
  }
}
```

窗体可以使用这个只读属性查看这个敌人在游戏地牢中是否可见。

各个Enemy子类都要实现这个方法。

玩家攻击一个敌人时，它会调用敌人的Hit()方法，这会将其点数减去一个随机数。

Enemy类继承了Mover的Nearby()方法，可以用来确定敌人是否靠近玩家。

如果将玩家的位置传入FindPlayerDirection()，它会使用基类的location字段来得出玩家相对于敌人的位置，并返回一个Direction enum，指出敌人需要沿哪个方向移动才能越来越靠近玩家。

# 编写不同的Enemy子类

这3个Enemy子类相当简单。每个敌人都有一个不同的起始点数，并以不同的方式移动，另外攻击时会造成不同程度的损伤。这3个子类要向Enemy基类构造函数分别传递一个不同的startingHitPoints参数，另外必须为各个子类编写不同的Move()方法。

下面给出一个例子，这是其中的一个类：

```
class Bat : Enemy {
    public Bat(Game game, Point location)
        : base(game, location, 6)
    { }

    public override void Move(Random random) {
        // Your code will go here
    }
}
```

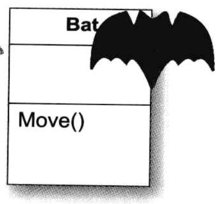
蝙蝠点数初始为6，所以向基类构造函数传入6。

这些子类构造函数中可能不需要任何操作，基类已经处理了全部事宜。

蝙蝠随机地飞来飞去，所以使用Random，从而一半情况下朝着随机的方向飞。

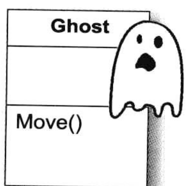
一旦敌人点数为0，窗体就不再显示这个敌人。但是它还在游戏的Enemies列表中，除非玩家结束这一关。

这些类派生自Enemy基类，而Enemy又派生自Mover。



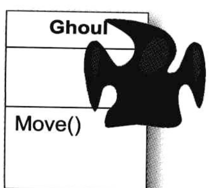

蝙蝠起始点数为6。只要点数等于或大于1，它会一直朝着玩家移动并发动攻击。移动时，有50%的几率朝着玩家飞，而另外50%的情况下会朝随机的方向飞。蝙蝠移动后，它会检查是否靠近玩家，如果确实很接近，则攻击玩家，使玩家点数最多下降2点。

必须保证窗体每一回合都要查看某个敌人是否可见。




幽灵比蝙蝠更难打败，不过与蝙蝠类似，只有当点数大于0时才能移动和攻击。它的起始点数为8点。移动时，有1/3的几率朝着玩家移动，另外2/3的情况下只是原地不动。如果它已经靠近玩家，就会攻击玩家，使玩家点数最多下降3点。

幽灵和食尸鬼都使用了Random，保证比玩家移动得慢。




食尸鬼是最可怕的敌人。它的起始点数为10点，同样地，只有点大于0时才能移动和攻击。移动时，有2/3的几率朝着玩家移动，另外1/3的情况下原地不动。如果靠近玩家，会向玩家发动攻击，造成玩家点数最多下降4点。

# Weapon继承自Mover，各种武器继承自Weapon

**Weapon (abstract)**
PickedUp
Location
PickUpWeapon()
DamageEnemy()

就像需要一个Enemy基类一样，我们还需要一个Weapon基类。每个武器都有一个位置，还有一个属性指示这个武器是否已选。下面给出Weapon基类：

Weapon继承自Mover，因为DamageEnemy()中使用了Mover的Nearby()和Move()方法。

```
abstract class Weapon : Mover {

  protected Game game;
  private bool pickedUp;
  public bool PickedUp { get { return pickedUp; } }
  private Point location;
  public Point Location { get { return location; } }

  public Weapon(Game game, Point location) {
    this.game = game;
    this.location = location;
    pickedUp = false;
  }

  public void PickUpWeapon() { pickedUp = true; }

  public abstract string Name { get; }

  public abstract void Attack(Direction direction, Random random);

  protected bool DamageEnemy(Direction direction, int radius,
                             int damage, Random random) {
    Point target = game.PlayerLocation;
    for (int distance = 0; distance < radius; distance++) {
      foreach (Enemy enemy in game.Enemies) {
        if (Nearby(enemy.Location, target, radius)) {
          enemy.Hit(damage, random);
          return true;
        }
      }
      target = Move(direction, target, game.Boundaries);
    }
    return false;
  }
}
```

已选的(pickedup)武器不应显示……窗体可以使用这个获取存取方法来确定武器是否已选。

每个武器在地牢中都有一个位置。

构造函数设置game和location字段，并设置pickedUp为false（因为还没有选择这个武器）。

各个武器类需要实现一个Name属性，并实现一个Attack()方法来确定武器如何攻击。

各个武器的Name属性返回武器名（"Sword"，"Mace"，"Bow"）。

每个武器的攻击范围和模式不同，所以各种武器以不同的方式实现Attack()方法。

Mover类中的Nearby()方法只有两个参数：一个Point和一个int，它将这个Point与Mover的location字段比较。需要增加一个重载Nearby()方法，这个方法基本上是相同的，只是要取3个参数：两个Point和一个距离，它要将第一个Point与第二个Point比较（而不是location字段）。

DamageEnemy()方法由Attack()调用。它尝试根据给定的方向和攻击半径寻找一个敌人。如果找到，则调用这个敌人的Hit()方法，并返回true。如果没有发现敌人，则返回false。

# 不同的武器以不同方式攻击

Weapon的各个子类有其自己的名字和攻击逻辑。你的任务是实现这些类。下面给出一个Weapon子类的基本骨架：

每个子类表示3种武器之一：宝剑（sword）、弓箭（bow）或重头棒（mace）。

```
class Sword : Weapon {

  public Sword(Game game, Point location)
      : base(game, location) { }

  public override string Name { get { return "Sword"; } }

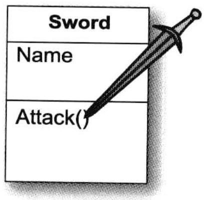
  public override void Attack(Direction direction, Random random) {
    // Your code goes here
  }
}
```

各子类依靠基类完成初始化工作。

各个武器的name属性中硬编码写入了武器名。

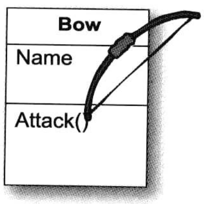
Game对象会传递攻击的方向。

玩家可以反复使用武器，不会丢掉或用完。

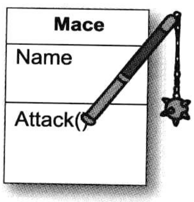



宝剑是玩家选择的第一个武器。它的攻击角度很大：攻击时，首先尝试攻击指定方向的敌人，如果这个方向没有敌人，则从原攻击方向顺时针旋转90度查看下一个方向，攻击这个方向上的敌人，如果还是没有找到敌人，则从原攻击方向逆时针旋转90度查看有没有敌人，并发动攻击。攻击半径是10，会导致敌人的点数损失3点。

仔细考虑这样一个问题……左方向再向右是什么方向，上方向再向左又是什么方向？




弓箭的攻击角度比较小，不过射程很长，攻击半径为30，不过只会导致敌人点数损失1点。宝剑会在3个方向发动攻击（因为玩家挥舞宝剑的弧度很大），不同于宝剑，玩家向某个方向射箭时，只会在这一个方向上射出。



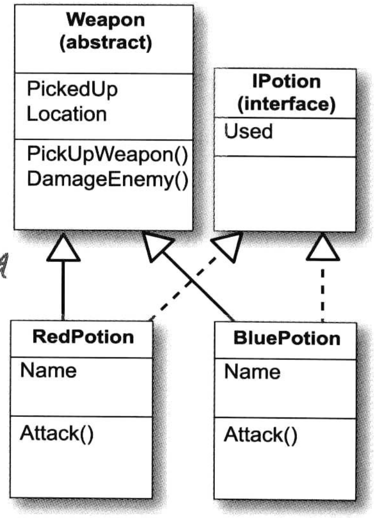

重头棒是地牢里最强大的武器。玩家在哪个方向上使用这个武器并不重要，因为它会360度全方位挥舞，它能攻击半径20以内的所有敌人，并导致点数下降6点。

不同武器以不同方式调用DamageEnemy()。Mace可以在各个方向发动攻击，所以如果玩家向右攻击，会先调用DamageEnemy(Direction.Right, 20, 6, random)。如果在这个方向上没有打到任何敌人，它会攻击Up方向。如果这里也没有敌人，会尝试Left，然后是Down，因此它能够360度全方位挥舞。

# Potion实现IPotion接口

共有两种补药，蓝色补药和红色补药，它们可以增加玩家的健康度。这两种补药就像武器一样，玩家可以在地牢中选择补药，可以点击武器清单里的补药作为装备，还可以通过点击某个攻击按钮来使用这些补药。所以让这两种补药继承抽象Weapon类是有道理的。

不过补药与其他武器还是存在一点区别的，所以需要增加一个IPotion接口，使它们还有另一个行为：增加玩家的健康度。IPotion接口相当简单。补药只需要增加一个只读属性，名为Used，如果玩家还没有用过这个补药则返回false，如果已经用过，这个属性就返回true。窗体将使用这个属性来确定是否在武器清单中显示这个补药。

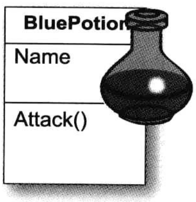



```
interface IPotion {
  bool Used { get; }
}
```

IPotion使补药只能使用一次。基于这个接口，还可以利用“if (weapon is IPotion)”来检查一个Weapon是否是补药。

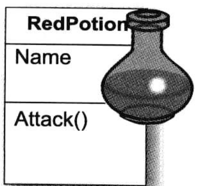
补药继承自Weapon类，因为它们的用法类似于武器，玩家在武器卷轴中点击补药可以立刻把它作为装备，然后点击任何攻击按钮就能使用了。

你应该能使用这个类图和以下信息编写这些类。




BluePotion类的Name属性应当返回串“Blue Potion”。玩家使用蓝色补药时就会调用其Attack()方法，它通过调用IncreasePlayerHealth()方法将玩家的健康度至多提升5点。玩家用完这个补药后，它的Used属性应当返回true。

如果玩家在第2关已经选择一个蓝色补药，可以使用这个补药，然后在第4关选择另一个蓝色补药，游戏会创建两个不同的BluePotion实例。




RedPotion类非常类似于BluePotion，只不过它的Name属性会返回串“Red Potion”，Attack()方法将把玩家的健康度至多提升10点。

## 窗体实现集成

这里有一个Game对象实例，它会作为窗体的一个私有字段。这个对象在窗体的Load事件中创建，窗体的各个事件处理程序通过使用Game对象的字段和方法保证游戏正常进行。

一切都从窗体的Load事件处理程序开始，它向Game传入一个Rectangle，定义了地牢游戏区的边界。以下窗体代码可以作为起点：

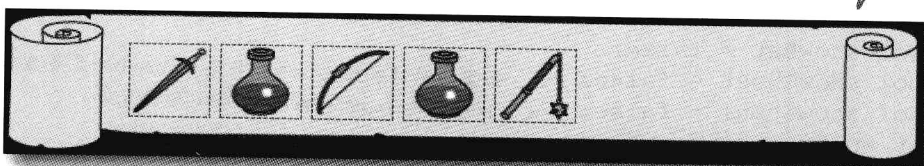
```
private Game game;
private Random random = new Random();
private void Form1_Load(object sender,
                        EventArgs e) {
    game = new Game(new Rectangle(78, 57, 420, 155));
    game.NewLevel(random);
    UpdateCharacters();
}
```

下载并向窗体增加背景图像，这是这个图像中地牢的边界。

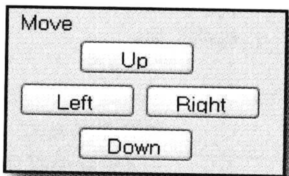
**使用Rectangle**

只要处理窗体就会看到大量Rectangle。可以通过传入X、Y、Width和Height值来创建一个Rectangle对象，也可以传入两个Point（表示两个对角点）来创建。得到一个Rectangle实例时，可以访问其Left、Right、Top和Bottom，也可以访问其X、Y、Width和Height值。

记住双击每一个PictureBox，使IDE为所有这些控件分别增加一个事件处理程序。

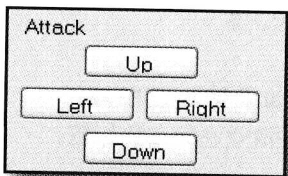


对应各个PictureBox的Click事件，窗体分别有一个事件处理程序。玩家点击宝剑时，首先使用Game对象的CheckPlayerInventory()方法来检查宝剑确实在玩家的武器清单中。如果玩家有宝剑，窗体再调用game.Equip()来装备这个武器。接下来设置各个PictureBox的BorderStyle属性，在宝剑四周画一个方框，并确保所有其他图标周围没有方框。




对应4个移动按钮分别有一个事件处理程序。它们都很简单。首先按钮调用game.Move()，并提供适当的Direction值，然后调用窗体的UpdateCharacters()方法。

玩家装备了宝剑、弓箭或重头棒时，要确保将按钮改回为原来的状态。




4个攻击按钮的事件处理程序也很简单。每个按钮调用game.Attack()，然后调用窗体的UpdateCharacters()方法。如果玩家装备的是补药，也同样需要调用game.Attack()，不过补药没有任何方向。所以玩家装备补药时，要让Left、Right和Down按钮不可见，并把Up按钮上的文本改为“Drink”。

# 窗体的UpdateCharacters()方法将PictureBoxes移到位

最后一个难点是窗体的UpdateCharacters()方法。一旦所有对象移动并相互作用，窗体就要完全更新……所以，对于已经选择的武器，要把相应PictureBox的Visible属性设置为false，另外要在新位置上显示敌人和玩家（如果已死，则不可见），还要更新武器清单。

你需要做到：

1. **更新玩家的位置和统计信息。**
   首先要更新玩家PictureBox的位置和显示点数的标签。需要一些变量来确定是否显示了各个敌人。

```
public void UpdateCharacters() {
  Player.Location = game.PlayerLocation;
  playerHitPoints.Text =
    game.PlayerHitPoints.ToString();

  bool showBat = false;
  bool showGhost = false;
  bool showGhoul = false;
  int enemiesShown = 0;
  // more code to go here...
```

如果让蝙蝠的PictureBox可见，showBat变量要设置为true。showGhost和showGhoul与此类似。

2. **更新各个敌人的位置和点数。**
   各个敌人可能在一个新位置，并有不同的点数。更新了玩家位置后还需要更新各个敌人的位置：

```
foreach (Enemy enemy in game.Enemies) {
  if (enemy is Bat) {
    bat.Location = enemy.Location;
    batHitPoints.Text = enemy.HitPoints.ToString();
    if (enemy.HitPoints > 0) {
      showBat = true;
      enemiesShown++;
    }
  }
  // etc...
```

这个代码就放在以上代码的下面。

这会影响敌人PictureBox控件的可见性。

在你的foreach循环中还需要另外两个像这样的if语句，一个对应幽灵，另一个对应食尸鬼。

一旦循环处理完这一关上的所有敌人，检查showBat变量。如果蝙蝠已经被杀死，showBat仍为false，因此将其PictureBox置为不可见，并清空它的点数标签。然后对showGhost和showGhoul做同样的处理。

**3** **更新武器PictureBox。**

声明一个weaponControl变量，并用一个switch语句设置它等于对应地牢中武器的PictureBox。

```
sword.Visible = false;
bow.Visible = false;
redPotion.Visible = false;
bluePotion.Visible = false;
mace.Visible = false;
Control weaponControl = null;
switch (game.WeaponInRoom.Name) {
   case "Sword":
      weaponControl = sword; break;
```

确保你的控件名与这些名一致。如果不一致，最后可能会出现一些很难调试的bug。

对应每种武器类型还需要其他case语句。

其他case语句应当把变量weaponControl设置为窗体中适当的控件。在switch语句后面，将weaponControl.Visible设置为true来显示这个控件。

**4** **设置各个武器清单图标PictureBox的Visible属性。**

使用Game对象的CheckPlayerInventory()方法确定是否显示各个武器清单图标。

**5** **以下是这个方法的其余部分。**

这个方法接下来做了三件事。首先查看玩家是否已经选择了地牢中的武器，从而知道是否要显示这个武器。然后查看玩家是否已死。最后查看玩家是否已经消灭所有敌人。如果是，则进入下一关。

```
weaponControl.Location = game.WeaponInRoom.Location;
if (game.WeaponInRoom.PickedUp) {
  weaponControl.Visible = false;
} else {
  weaponControl.Visible = true;
}
if (game.PlayerHitPoints <= 0) {
  MessageBox.Show("You died");
  Application.Exit();
}
if (enemiesShown < 1) {
  MessageBox.Show("You have defeated the enemies on this level");
  game.NewLevel(random);
  UpdateCharacters();
}
```

每一关都有一个武器。如果已经选择，要让这个武器的图标不可见。

Application.Exit()立即退出程序。这是System.Windows.Forms的方法，所以如果希望在窗体之外使用这个方法，需要有相应的using语句。

如果这一关上再没有敌人，说明玩家已经把它们都消灭，可以进入下一关。

# 好戏就要开始了!

7重难关，3个敌人…… 这个游戏已经像模像样了。不过你还可以让它更棒。下面是几点建议供你参考……

## 让敌人更聪明

如何修改敌人的Move()方法，使它们更难打败？然后看看能不能把常量改为敌人的属性，并且提供一种办法在游戏中改变这些属性。

## 增加更多关

这个游戏不一定到7关就结束。你看能不能增加更多关……有没有办法让游戏一直运行下去？如果玩家确实赢了，做一个很酷的结束动画，让幽灵和蝙蝠为玩家跳舞！如果玩家死了，游戏就会立即结束。你能想出一个更友好的结束方式吗？也许可以让用户重新开始游戏或者从他的最后一关再来一次。

## 增加不同种类的敌人

不需要限制危险只来自食尸鬼、幽灵和蝙蝠。看看能不能为这个游戏增加更多敌人。

## 增加更多武器

玩家非常需要更多帮助来打败你增加的新敌人。想想武器还能采用哪些方式进行攻击，或者补药还能做什么。另外由于Weapon是Mover的一个子类，可以利用这一点让玩家去追这些魔力的武器!

## 增加更多图片

可以访问www.headfirstlabs.com/books/hfcsharp/为新增的敌人和武器找到更多图像文件，充分展现你的想象力。

## 让它成为一个动作游戏

这是一个有意思的挑战。你能想办法使用第4章按键游戏中的KeyDown事件和Timer把这个基于回合的游戏变成一个动作游戏吗？

**你可以借这个机会露一手！如果你为这个游戏开发了一个很酷的新版本，可以加入《Head First C#》论坛（www.headfirstlabs.com/books/hfcsharp/）炫耀一番！**

# 9 读写文件

# 保存了字节数组，就留住了一切




**有时多点持久性是有回报的。**

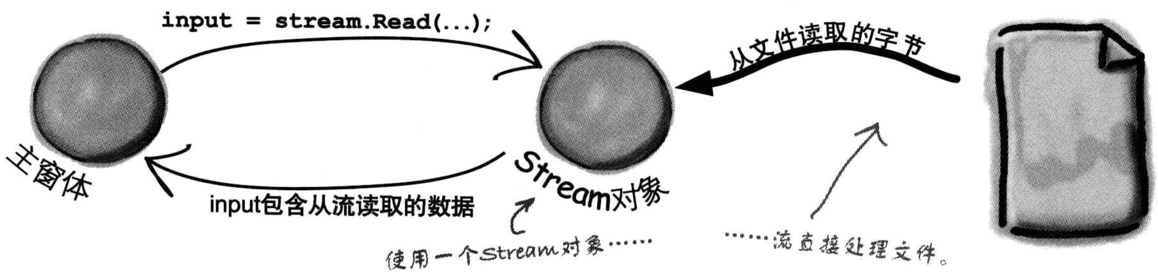
到目前为止，所有程序的生命都很短暂。它们启动后，运行一段时间，然后就会关闭。但这往往不够（特别是处理非常重要的信息时）。需要能够把你的工作保存下来。在这一章中，我们将介绍如何向文件写数据，然后如何从文件读回这个信息。你将了解.NET流类，还会掌握十六进制和二进制的一些奥秘。

# .NET使用流读写数据

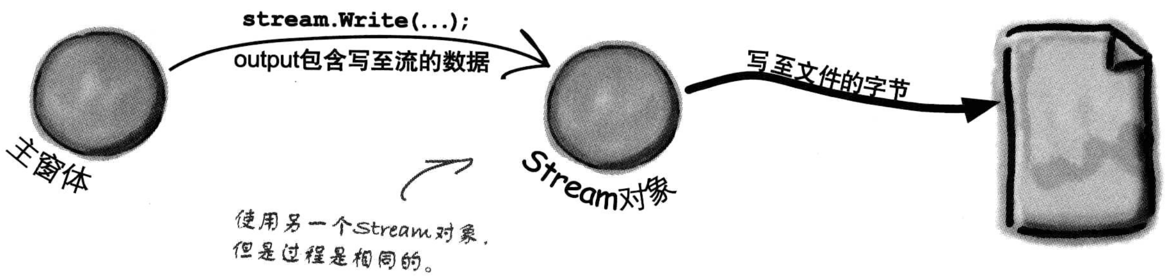
流（stream）是.NET Framework为程序读写数据提供的方法。只要程序读写一个文件，连接网络上的另一个计算机，或者更一般地，从一处向另一处发送或接收数据字节，就是在使用流。

**如果希望对文件读写数据，就会使用一个Stream对象。**

**假设有一个简单的程序,窗体有一个事件处理程序需要从一个文件读取数据。可以使用一个Stream对象来完成这个工作。**

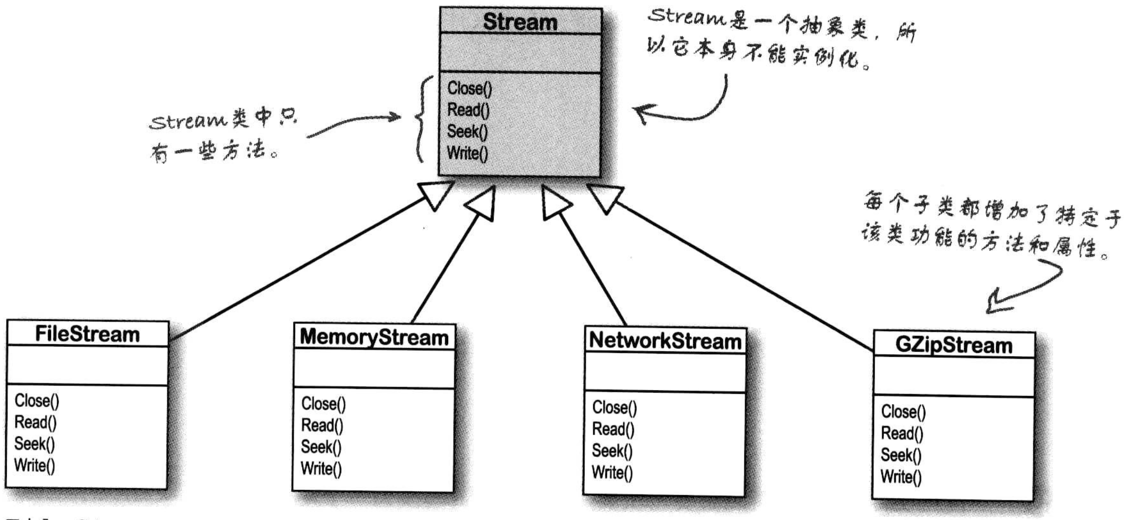



**如果程序需要将数据写到文件，可以使用另一个Stream对象。**



# 不同的流读写不同的数据

每个流都是抽象类Stream的子类，已经提供了很多内置流类，它们可以完成不同的工作。我们将重点介绍常规文件的读写，不过这一章学到的所有知识同样可以很容易地应用到压缩文件或加密文件，也可以应用到根本不使用文件的网络流。




`FileStreams`允许读写文件。

`MemoryStreams`允许向内存块读写数据。

`NetworkStream`对象允许向网络上的其他计算机或设备读写数据。

`GZipStream`允许压缩数据，从而占更少空间，并且更易于下载和存储。

## 流能做的事情：

1. **写流。**
   通过流的`Write()`方法可以把数据写至一个流。

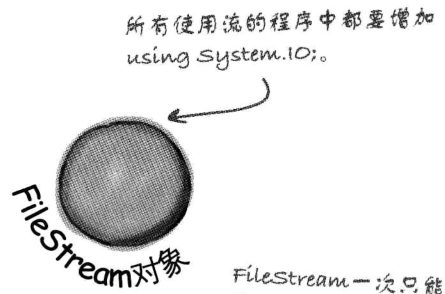
2. **读流。**
   使用流可以利用`Read()`方法从文件、网络、内存或任何地方获取数据。

3. **改变在流中的位置。**
   大多数流都支持一个`Seek()`方法，允许查找流中的一个位置，从而能够在特定位置插入数据。

**流允许读写数据。要针对所处理的数据使用适当的流。**

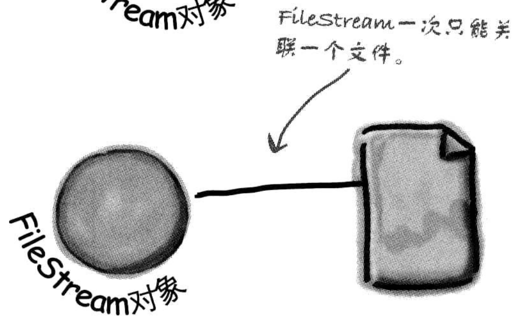
# FileStream向文件读写字节

程序要向文件写几行文本时，必须做很多事情。




1. 创建一个新的FileStream对象，通知它写文件。

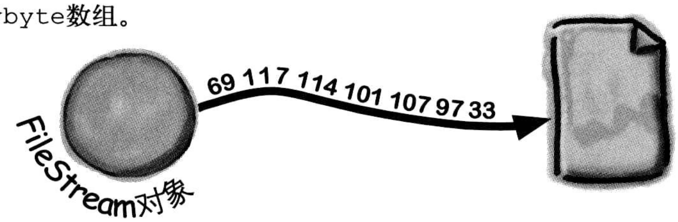
2. FileStream将自己关联到一个文件。



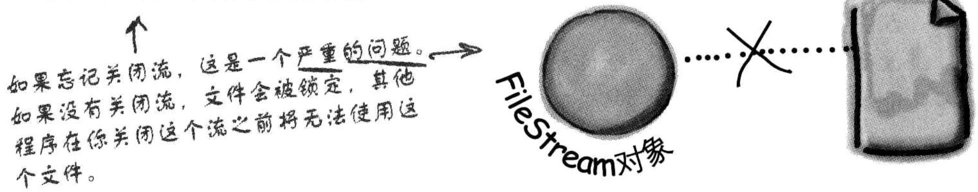

3. 流向文件写字节，所以需要将所要写的strubg转换为一个byte数组。




4. 调用流的Write()方法，并传入这个byte数组。




5. 关闭文件，使其他程序能够访问这个文件。



# 只用3个简单步骤就能向文件写文本

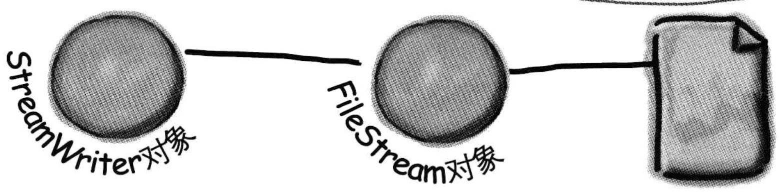
**StreamWriter会自动创建并管理FileStream对象。**

C#提供了一个便利类，名为StreamWriter，它只用一步就能完成以上所有工作。你要做的就是创建一个新的StreamWriter对象，并提供一个文件名。它会自动地创建一个FileStream，并打开这个文件。然后可以使用StreamWriter的Write()和WriteLine()方法向文件写入你想写的任何内容。

**1 使用StreamWriter的构造函数打开或创建一个文件。**

可以向StreamWriter()构造函数传入一个文件名。如果传入了文件名，书写器（writer）会自动打开这个文件。StreamWriter的另一个重载构造函数还有一个bool参数，如果希望向一个现有文件的末尾增加文本（或追加），这个参数就为true，如果想删除现有文件，而创建一个同名的新文件，这个参数则为false。

```
StreamWriter writer = new StreamWriter(@"C:\newfiles\toaster oven.txt", true);
```

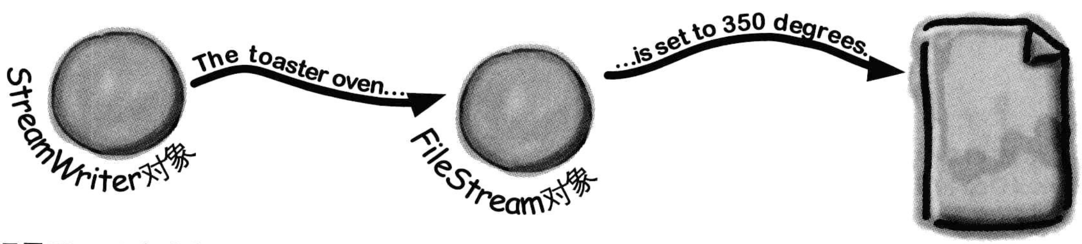



在文件名前面加@就是在告诉C#把它当作一个字面量串，而不要对字符转义（如\t转义为制表符，或\n转义为换行符）。

**2 使用Write()和WriteLine()方法写文件。**

这些方法的工作与Console中的相应方法是类似的：Write()写文本，WriteLine()写文本并在最后增加一个换行符。如果所写的串中包含“{0}”、“{1}”、“{2}”等，Write()和WriteLine()方法会在写出的串中包含参数：“{0}”替换为所写串后面的第一个参数，“{1}”替换为第二个参数，依此类推。

```
writer.WriteLine("The {0} is set to {1} degrees.", appliance, temp);
```




**3 调用Close()方法释放文件。**

如果流关联到一个文件，让流一直打开的话就会将这个文件锁定，其他程序将无法使用这个文件。所以一定要关闭文件！

```
writer.Close();
```

# 恶魔Swindler又要发动一个邪恶计划

Objectville的居民们一直生活在恶魔Swindler的阴影之下。现在他要使用一个StreamWriter来实现另一个邪恶计划。下面来看发生了什么。创建一个新的控制台应用，为Main()方法增加以下代码：

写到根文件夹可能不是一个好想法，你的操作系统可能甚至不允许这么做。所以另选一个要写入文件的文件夹。

这一行创建一个StreamWriter对象，并告诉它文件在哪里。

路径以一个@符号开头，所以StreamWriter不会把“\”解释为一个转义序列的开始。

```
StreamWriter sw = new StreamWriter(@"C:\secret_plan.txt");
sw.WriteLine("How I'll defeat Captain Amazing");
sw.WriteLine("Another genius secret plan by The Swindler");
sw.Write("I'll create an army of clones and ");
sw.WriteLine("unleash them upon the citizens of Objectville.");
string location = "the mall";
for (int number = 0; number <= 6; number++){
    sw.WriteLine("Clone #{0} attacks {1}", number, location);
    if (location == "the mall") { location = "downtown"; }
    else { location = "the mall"; }
}
sw.Close();
```

WriteLine()写文本之后会增加一个换行符。Write()只发送文本，不会在最后另外增加换行。

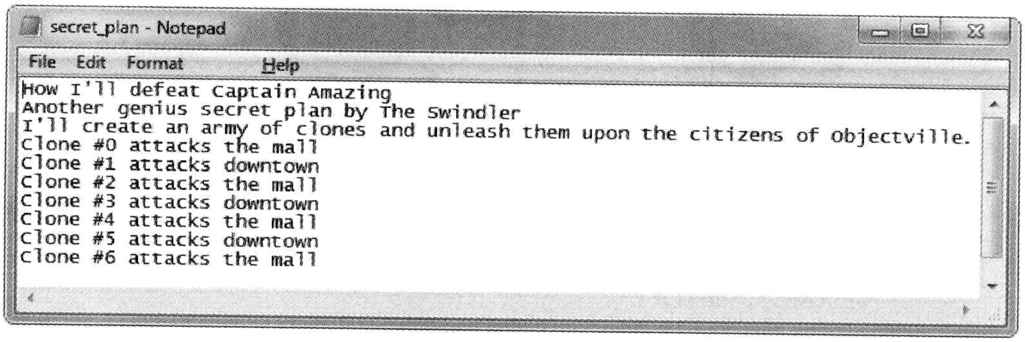
你能看出这个代码中的location变量会有什么变化吗？

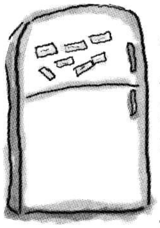
可以在文本中使用{}将变量传入所写的串。{0}会替换为串后面的第一个参数，{1}替换为第二个参数，依此类推。

Close()释放这个文件的所有连接，以及StreamWriter使用的所有资源。如果没有关闭流，文本就不会写入文件。

这是以上代码生成的输出。

**StreamWriter在System.IO命名空间中，所以一定要在程序最前面增加“using System.IO;”。**





# StreamWriter磁条

假设有以下button1_Click()代码。你的任务是使用这些磁条建立Flobbo类的代码，使得调用这个事件处理程序时，它会生成这一页最下面所示的输出。祝你好运！

```
private void button1_Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

```
sw.WriteLine(Zap);
Zap = "red orange";
return true;
```

```
}
```

```
sw.WriteLine(Zap);
sw.Close();
return false;
```

```
public bool Blobbo
    (bool Already, StreamWriter sw) {
```

```
public bool Blobbo(StreamWriter sw) {
```

```
sw.WriteLine(Zap);
Zap = "green purple";
return false;
```

```
}
```

```
return new
  StreamWriter("macaw.txt");
```

```
}
```

```
}
```

```
}
```

```
private string Zap;

public Flobbo(string Zap) {
    this.Zap = Zap;
}
```

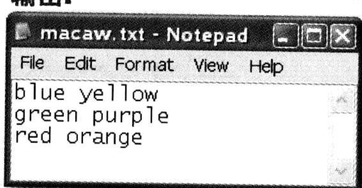
```
class Flobbo {
```

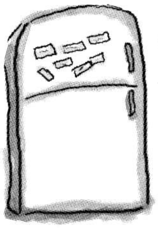
```
if (Already) {
```

```
} else {
```

```
public StreamWriter Snobbo() {
```

**输出：**





# StreamWriter磁条答案

你的任务是利用磁条构造Flobbo类来得到所要的输出。

```
private void button1 _ Click(object sender, EventArgs e) {
    Flobbo f = new Flobbo("blue yellow");
    StreamWriter sw = f.Snobbo();
    f.Blobbo(f.Blobbo(f.Blobbo(sw), sw), sw);
}
```

提醒：在这个题目里我们特意选择了一些奇怪的变量名和方法，因为如果我们选用的名字很好，这个题目就太容易了！不过，在你自己的代码里不要使用这种名字，好吗？

```
class Flobbo {
    private string Zap;

    public Flobbo(string Zap) {
        this.Zap = Zap;
    }

    public StreamWriter Snobbo() {
        return new
          StreamWriter("macaw.txt");
    }

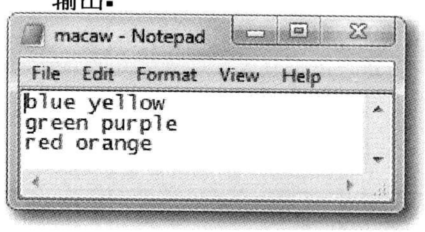
    public bool Blobbo(StreamWriter sw) {
        sw.WriteLine(Zap);
        Zap = "green purple";
        return false;
    }

    public bool Blobbo
        (bool Already, StreamWriter sw) {
        if (Already) {
            sw.WriteLine(Zap);
            sw.Close();
            return false;
        } else {
            sw.WriteLine(Zap);
            Zap = "red orange";
            return true;
        }
    }
}
```

Blobbo()方法是重载的，它的两个声明有不同的参数。

确保用完后关闭文件。

输出：



# 使用两个对象读写

> **简单提示。这里的"stream"有些含糊。StreamReader（继承自TextReader）是一个从流读取字符的类。它本身不是一个流。将一个文件名传入这个类的构造函数时，它会为你创建一个流，当你调用Close()方法时它会关闭这个流。StreamReader还有一个重载的构造函数，取一个Stream为参数。可以自己看看它是如何做的？**

下面用另一个流StreamReader读入Swindler的秘密计划，StreamReader的工作类似于StreamWriter，只不过并不是写文件，而要把所读文件的文件名提供给阅读器（reader）的构造函数。ReadLine()方法返回一个串，其中包含从文件读取的一行文本。可以写一个循环读取文件中每一行，直到其EndOfStream字段为true，此时已经读完文件中的所有文本行：

```
StreamReader reader =
   new StreamReader(@"c:\secret _ plan.txt");
StreamWriter writer =
   new StreamWriter(@"c:\emailToCaptainAmazing.txt");
writer.WriteLine("To: CaptainAmazing@objectville.net");
writer.WriteLine("From: Commissioner@objectiville.net");
writer.WriteLine("Subject: Can you save the day... again?");
writer.WriteLine();
writer.WriteLine("We've discovered the Swindler's plan:");
while (!reader.EndOfStream) {
  string lineFromThePlan = reader.ReadLine();
  writer.WriteLine("The plan -> " + lineFromThePlan);
}
writer.WriteLine();
writer.WriteLine("Can you help us?");
writer.Close();
reader.Close();
```

将要读的文件传入StreamReader的构造函数。

这个程序使用一个StreamReader来读取Swindler的计划，并使用一个StreamWriter写一个文件，这个文件将通过email发送给Captain Amazing。

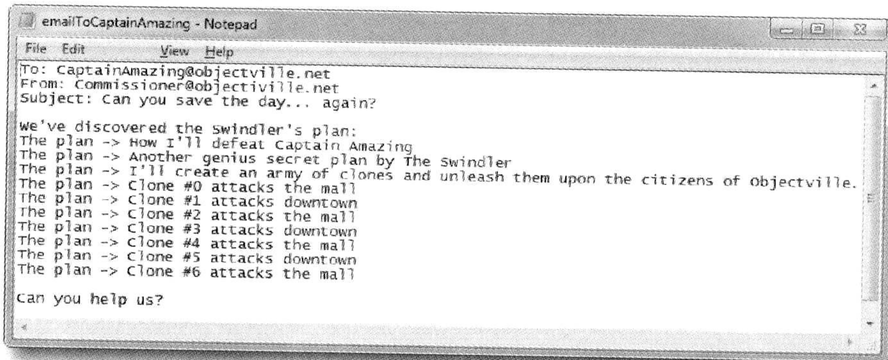
不带参数的WriteLine()方法写一个空行。

EndOfStream是一个属性，指出文件中是否还有未读的数据。

这个循环从阅读器读一行，并写到书写器。

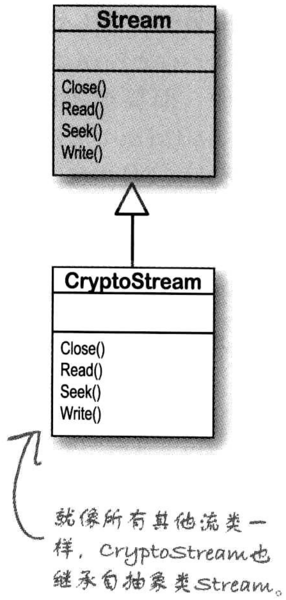
确保关闭所有已经打开的流，即使只是用来读文件的流也要关闭。

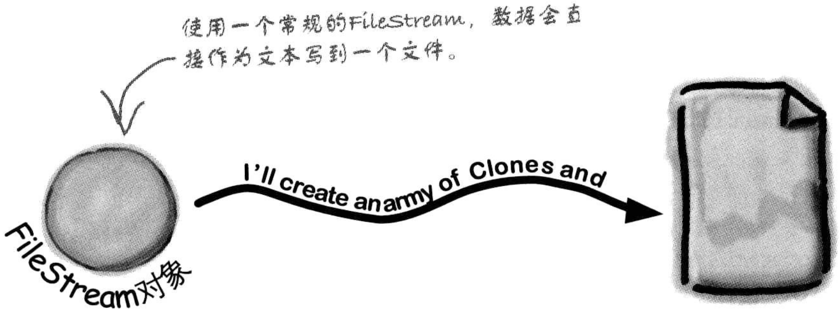
实例化时，StreamReader和StreamWriter会打开自己的流。调用它们的Close()方法会告诉它们关闭这些流。

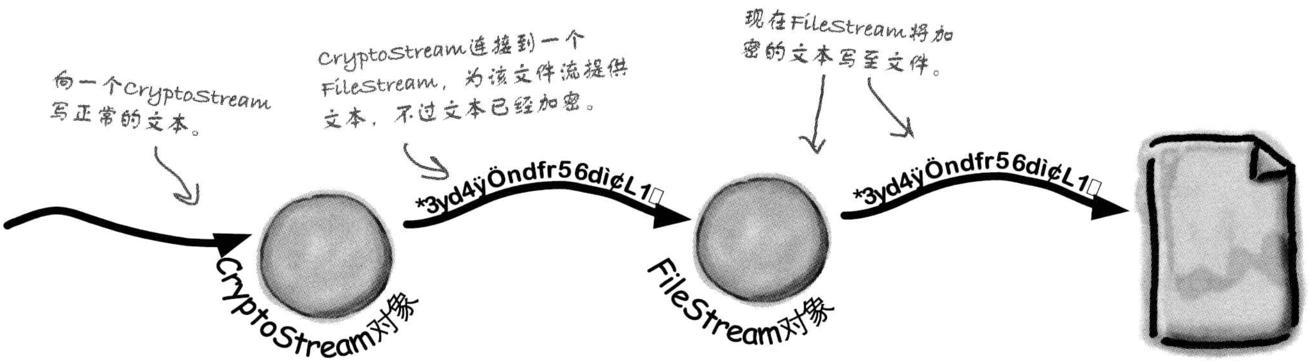


# 数据可以经过多个流

.NET中使用流的一大好处是，数据到达其最终目的地之前可以经过多个流。.NET内置了很多不同类型的流，其中有一个CryptoStream类。在做其他处理之前，可以利用这个类先对数据加密：



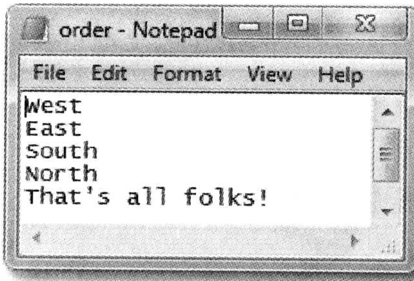




**可以把流串起来。一个流可以写到另一个流，而后者又可以再写到下一个流……最后通常是网络或一个文件流。**

# 池塘谜题



你的任务是从池塘里取出代码片段，在程序中填空。一个代码片段可以使用多次，并非所有片段都要用到。你的目标是建立一个程序来生成右边所示的输出。

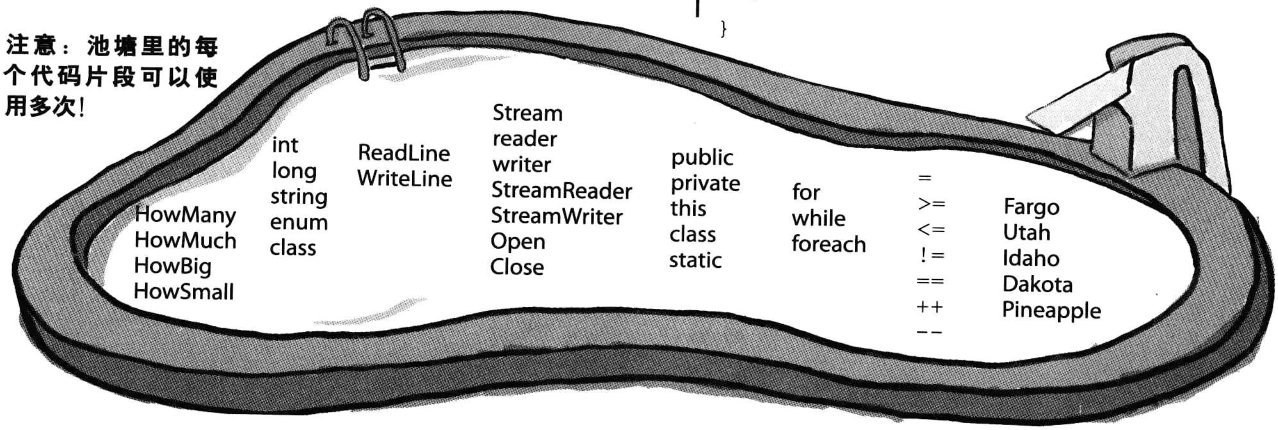



```
class Pineapple {
  const ______ d = "delivery.txt";
  public ______ ______
      { North, South, East, West, Flamingo }
  public static void Main() {
    __________ o = new ____________("order.txt");
    Pizza pz = new Pizza(new __________(d, true));
    pz.________(Fargo.Flamingo);
    for (_____ w = 3; w >= 0; w--) {
      Pizza i = new Pizza
          (new ___________(d, false));
      i.Idaho((Fargo)w);
      Party p = new Party(new __________(d));
      p.___________(o);
    }
    o.___________("That's all folks!");
    o.__________();
  }
}
```

```
class Pizza {
  private ____________ ________;
  public Pizza(___________ _________) {
    ______.writer = writer;
  }
  public void ______(_______.Fargo f) {
    writer.__________(f);
    writer.__________();
  }
}

class Party {
  private ____________ reader;
  public Party(____________ reader) {
    ___________.reader = reader;
  }
  public void HowMuch(__________ q) {
    q.__________(reader.__________());
    reader.__________();
  }
}
```

**注意：池塘里的每个代码片段可以使用多次！**





# 池塘谜题答案

这个enum（具体来讲，实际上是它的ToString()方法）将用来打印大量输出。

这是程序的入口点。它创建一个StreamWriter，传入Pizza类。然后循环处理Fargo的成员，分别传入Pizza.Idaho()方法进行打印。

```
class Pineapple {
    const string d = "delivery.txt";
    public enum Fargo { North, South, East, West, Flamingo }
    public static void Main() {
        StreamWriter o = new StreamWriter("order.txt");
        Pizza pz = new Pizza(new StreamWriter(d, true));
        pz.Idaho(Fargo.Flamingo);
        for (int w = 3; w >= 0; w--) {
            Pizza i = new Pizza(new StreamWriter(d, false));
            i.Idaho((Fargo)w);
            Party p = new Party(new StreamReader(d));
            p.HowMuch(o);
        }
        o.WriteLine("That's all folks!");
        o.Close();
    }
}
```

Pizza类有一个StreamWriter作为私有字段，它的Idaho()方法使用Fargo enum的ToString()方法（由WriteLine()自动调用）将Fargo enum写至文件。

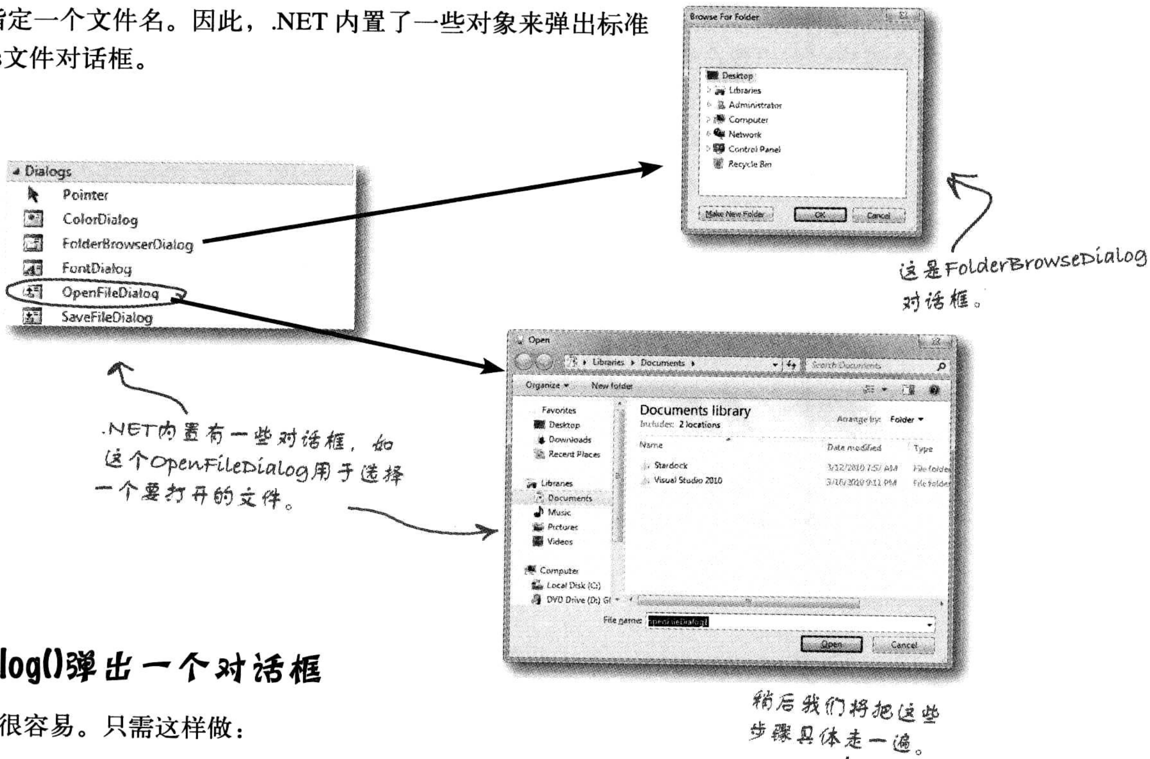
```
class Pizza {
    private StreamWriter writer;
    public Pizza(StreamWriter writer) {
        this.writer = writer;
    }
    public void Idaho(Pineapple.Fargo f) {
        writer.WriteLine(f);
        writer.Close();
    }
}
```

Party类有一个StreamReader字段，它的HowMuch()方法从这个StreamReader读入一行，并写至一个StreamWriter。

```
class Party {
    private StreamReader reader;
    public Party(StreamReader reader) {
        this.reader = reader;
    }
    public void HowMuch(StreamWriter q) {
        q.WriteLine(reader.ReadLine());
        reader.Close();
    }
}
```

# 使用内置对象弹出标准对话框

编写一个读写文件的程序时，往往需要在某个时刻弹出一个对话框提示用户指定一个文件名。因此，.NET 内置了一些对象来弹出标准的Windows文件对话框。




## ShowDialog()弹出一个对话框

显示对话框很容易。只需这样做：

1. 创建对话框对象的一个实例。可以在代码中使用`new`来创建，也可以从工具条把对话框控件拖到窗体上。

2. 设置对话框对象的属性。有一些很有用的属性，如`Title`（设置标题栏中的文本）、`InitialDirectory`（指示首先打开哪个目录），以及`FileName`（打开和保存对话框要用到这个属性）。

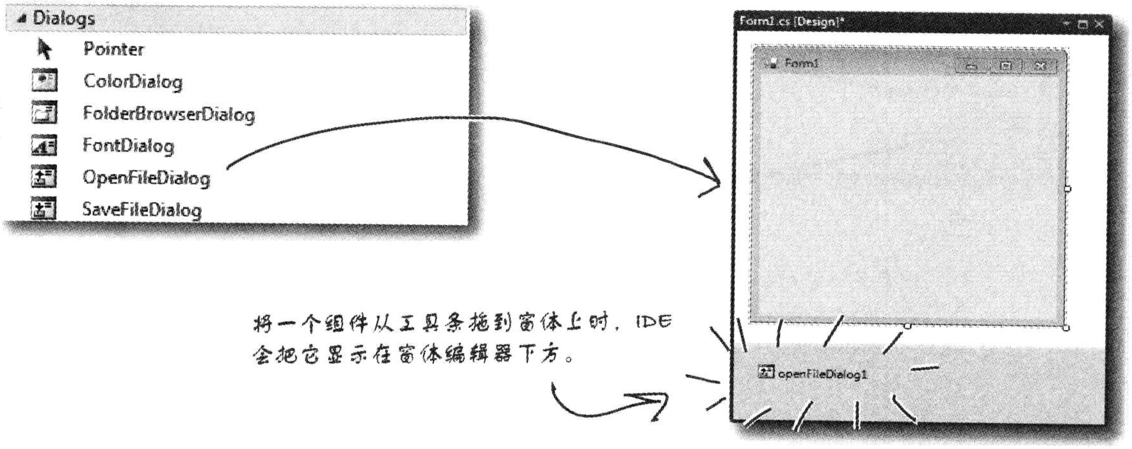
3. 调用对象的`ShowDialog()`方法。这会弹出对话框，直到用户点击OK按钮或Cancel按钮，或者关闭了窗口，否则对话框不会返回。

4. `ShowDialog()`方法返回一个`DialogResult`，这是一个enum。其成员包括OK（表示用户点击了OK），`Cancel`，`Yes`和`No`（用于`Yes/No`对话框）。

# 对话框也是.NET控件

只需把对话框控件拖到窗体上，就可以为你的程序增加Windows标准文件对话框。将一个OpenFileDialog控件从工具条拖出，放在窗体上。此时它不会作为一个可视化控件显示，你会看到它将出现在窗体下方。这是因为它是一个组件，所谓组件（component）是一种特殊的非可视化Toolbox控件，不会在窗体上直接显示，但是就像使用其他控件一样,可以在窗体的代码中使用这些组件。

"非可视化"只是表示：从工具条拖出后它不会出现在窗体上。




InitialDirectory属性会改变对话框打开时最先显示的文件夹。

通过Filter属性，可以改变对话框最下面显示的过滤器，比如要显示哪些类型的文件。

```
openFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";
openFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";
openFileDialog1.FileName = "default _ file.txt";
openFileDialog1.CheckFileExists = true;
openFileDialog1.CheckPathExists = false;
DialogResult result = openFileDialog1.ShowDialog();
if (result == DialogResult.OK){
    OpenSomeFile(openFileDialog1.FileName);
}
```

这些属性告诉对话框如果用户试图打开硬盘上一个不存在的文件或路径，要显示一个错误消息。

使用ShowDialog()方法显示对话框，会返回一个DialogResult。这是一个enum，可以用来检查用户是否点击OK按钮。如果用户点击了OK，则会设置为DialogResult.OK，如果用户点击了Cancel，将设置为DialogResult.Cancel。

# 对话框也是对象

**OpenFileDialog**对话框对象会显示标准的Windows“打开”窗口，**SaveFileDialog**会显示“保存”窗口。要显示这些对话框，可以创建一个新实例，设置对象的属性，并调用`ShowDialog()`方法。`ShowDialog()`方法返回一个`DialogResult` enum（因为有些对话框有多个按钮或结果，所以只使用bool是不够的）。

```
saveFileDialog1 = new SaveFileDialog();
saveFileDialog1.InitialDirectory = @"c:\MyFolder\Default\";
saveFileDialog1.Filter = "Text Files (*.txt)|*.txt|"
  + "Comma-Delimited Files (*.csv)|*.csv|All Files (*.*)|*.*";
DialogResult result = saveFileDialog1.ShowDialog();
if (result == DialogResult.OK){
    SaveTheFile(saveFileDialog1.FileName);
}
```

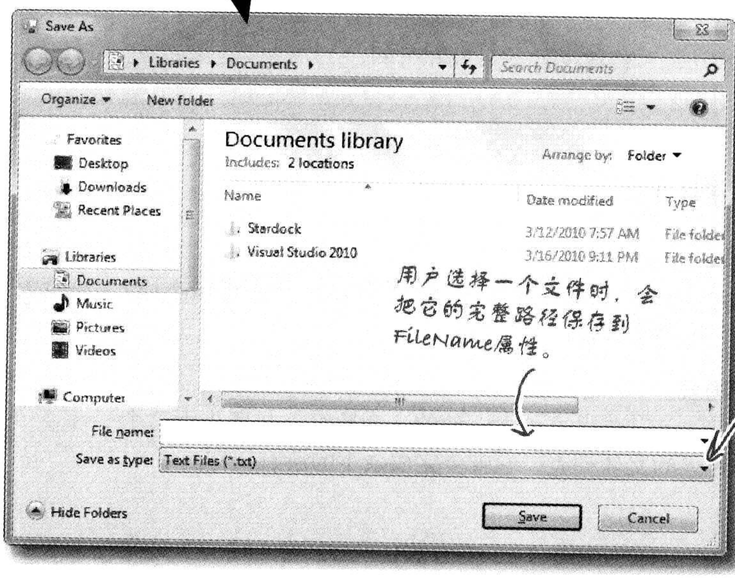
将一个保存对话框对象拖出工具条放到窗体上时，IDE只会向窗体的InitializeComponent()方法增加这样一行代码。

Filter属性不难理解。只需将串中“|”字符之间的内容与窗口中显示的内容做个比较。

ShowDialog()和FileName属性与OpenFileDialog对象中相应方法的工作完全相同。

SaveFileDialog对话框弹出标准的Windows“另存为…”对话框。

通过Title属性可以改变这个文本。




用户选择一个文件时，会把它的完整路径保存到FileName属性。

ShowDialog()方法弹出对话框，并打开InitialFolder属性中指定的文件夹。

使用Filter属性改变“Save as type”列表。

利用ShowDialog()方法返回的DialogResult，可以确定用户点击了哪个按钮。

# 使用内置File和Directory类处理文件和目录

类似于StreamWriter，File类也会创建流，以便你在后台处理文件。可以使用这个类的方法完成大多数常见的操作，而不需要先创建FileStream。利用Directory对象，可以处理包含文件的整个目录。

## 使用File可以做到：

1. 查找文件是否存在。可以使用Exists()文件检查一个文件是否存在。如果这个文件存在，则返回true，如果不存在则返回false。

2. 读写文件。
可以使用OpenRead()方法从文件获取数据，或者使用Create()或OpenWrite()方法写文件。

3. 向文件追加文本。
AppendAllText()方法允许你向一个已经创建的文件追加文本。如果运行这个方法时文件并不存在，则会创建这个文件。

4. 获得文件的有关信息。
GetLastAccessTime()和GetLastWriteTime()方法能返回最后一次访问和修改文件的日期和时间。

**FileInfo类似于File**

如果希望对文件做很多操作，也许需要创建FileInfo类的一个实例而不是使用File类的静态方法。

File类能做的所有工作FileInfo类都能完成，只不过FileInfo类必须先实例化才能使用。可以创建FileInfo的一个新实例，并以同样的方式访问其Exists()方法或OpenRead()方法。

唯一的区别是，File类完成少量操作时速度更快，而FileInfo更适合完成大量操作。

File是一个静态类，所以它只是一组可以用来处理文件的方法。FileInfo是一个可以实例化的对象，其方法与File中的方法完全相同。

## 使用Directory可以做到：

1. **创建一个新目录。**
使用CreateDirectory()方法可以创建一个新目录。只需提供路径，这个方法会负责余下的所有工作。

2. **获得目录中的文件列表。**
使用GetFiles()方法可以获得目录中文件的一个数组，只要告诉这个方法你想得到哪个目录的文件列表，它就会完成余下的所有工作。

3. **删除目录。**
删除目录也相当简单。只需使用Delete()方法。

## there are no Dumb Questions

**问：我还是不清楚StreamWriter里{0}和{1}是做什么的。**

**答：** 向文件输出串时，你会发现经常需要输出很多变量的内容。例如，可能必须写以下代码：

```
writer.WriteLine("My name is " + name +
    "and my age is " + age);
```

如果一直使用+来连接串，这会很麻烦，而且也容易出错。更容易的做法是利用{0}和{1}：

```
writer.WriteLine(
  "My name is {0} and my age is {1}",
   name, age);
```

读这个代码也会容易得多，特别是如果同一行中包含多个变量时，更能体现出这种代码的可读性。.

**问：为什么要在包含文件名的串前面放一个@？**

**答：** 向程序提加一个字符串字面量时，编译器会把类似\n和\r的转义序列转换为特殊字符。这样一来，键入文件名会很困难，因为文件名里有很多反斜线字符。如果在一个串前面加上@，它会告诉 C#不要把它解释为转义序列。另外还可以告诉C#在串中包含换行符，因此你可以在字符串中间按回车，它将作为一个换行符出现在输出里：

```
string twoLine = @"this is a string
that spans two lines.";
```

**问：\n 和 \t 又是什么意思？**

**答：** 它们是转义序列。\n是一个换行符，\t 是一个tab制表符。\r是回车，或者是Windows回车符的一半，在Windows文本文件中，行以\r\n结尾（第8章介绍Environment.NewLine时曾讨论过）。如果确实想在串中使用反斜线，不希望C#把它解释为一个转义序列的开始，只需要使用两个反斜线\\。

**问：把串转换为字节数组的初衷是什么？到底是怎么做到的？**

**答：** 你可能已经多次听说过，磁盘上的文件表示为二进制位和字节。这是说，向磁盘写一个文件时，操作系统会把它看作是一个很长的字节序列。StreamReader和StreamWriter会为你将字节转换为字符—这称为编码和解码。还记得第4章曾说过，byte变量可以存储0到255之间的任何数字。硬盘上的每个文件就是由0到255之间的数字组成的一个长序列。读写这些文件的程序要把这些字节解释为有意义的数据。在记事本程序中打开一个文件时，它会把各个单个的字节转换为一个字符，例如，E是69，a是97（不过这取决于具体的编码方式……稍后会了解更多有关编码的内容）。在记事本中键入文本并保存时，记事本会把各个字符再转换回字节，并保存到磁盘上。如果想向流写一个string，也要做同样的处理。

**问：如果我只是使用一个StreamWriter来写文件，又何必关心它是否为我创建了一个FileStream呢？**

**答：** 如果只是按部就班地向一个文本文件读写文本行，那么只需要StreamReader和StreamWriter。但是如果需要做更复杂的工作，那就还要使用其他的流。如果需要将数字、数组、集合或对象等数据写入一个文件，StreamWriter是做不到的。但是不用担心，稍后我们就会更详细地介绍如何实现。

**问：如果我想创建自己的对话框呢？怎么创建？**

**答：** 当然可以。可以向你的工程增加一个新的窗体，按你的想法来设计这个窗体。再用new创建它的一个新实例（就像创建OpenFileDialog对象一样）。然后可以调用它的ShowDialog()方法，它就能像所有其他对话框一样工作了。我们将在第13章更详细地讨论如何向程序中增加其他窗体。

**问：为什么用完之后要把流关闭？**

**答：** 你有没有遇到过这种情况：字处理程序告诉你它无法打开一个文件，因为这个文件"很忙"？一个程序在使用一个文件时，Windows会把这个文件锁定，不允许其他程序使用。程序打开一个文件时也是如此。如果没有调用Close()方法，你的程序就有可能将一个文件锁定，它会一直打开，直到程序结束。

## Sharpen your pencil

.NET有两个内置类，提供了大量静态方法来处理文件和文件夹。**File**类提供了处理文件的方法，另外可以使用**Directory**类处理目录。你认为以下各行代码分别有什么作用，请写下来。

| 代码 | 代码的作用 |
|---|---|
| `if (!Directory.Exists(@"c:\SYP")) {`<br>`    Directory.CreateDirectory(@"c:\SYP");`<br>`}` | |
| `if (Directory.Exists(@"c:\SYP\Bonk")) {`<br>`    Directory.Delete(@"c:\SYP\Bonk");`<br>`}` | |
| `Directory.CreateDirectory(@"c:\SYP\Bonk");` | |
| `Directory.SetCreationTime(@"c:\SYP\Bonk",`<br>`        new DateTime(1976, 09, 25));` | |
| `string[] files = Directory.GetFiles(@"c:\windows\",`<br>`        "*.log", SearchOption.AllDirectories);` | |
| `File.WriteAllText(@"c:\SYP\Bonk\weirdo.txt",`<br>`        @"This is the first line`<br>`and this is the second line`<br>`and this is the last line");` | |
| `File.Encrypt(@"c:\SYP\Bonk\weirdo.txt");`<br>看看能不能猜出这个代码做什么，这是你以前没见过的。 | |
| `File.Copy(@"c:\SYP\Bonk\weirdo.txt",`<br>`        @"c:\SYP\copy.txt");` | |
| `DateTime myTime =`<br>`        Directory.GetCreationTime(@"c:\SYP\Bonk");` | |
| `File.SetLastWriteTime(@"c:\SYP\copy.txt", myTime);` | |
| `File.Delete(@"c:\SYP\Bonk\weirdo.txt");` | |

# 使用文件对话框打开和保存文件（只需几行代码而已）

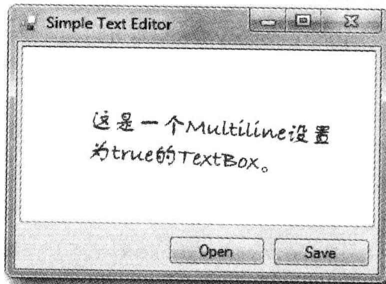
可以构建一个打开文本文件的程序。利用这个程序，可以修改这个文件，并保存所做的修改。为此只需很少的代码，一切都靠标准.NET控件来完成。步骤如下：

动手做！

下面告诉你一个让TextBox填满窗体的技巧。从Containers工具箱把TableLayoutPanel拖到窗体上，将它的Dock属性设置为Fill，使用它的Rows和Columns属性编辑器设定有两行一列。将TextBox拖到最上面的单元格中，然后从Toolbox将一个FlowLayoutPanel拖到下面的单元格，将其Dock设置为Fill,FlowDirection属性设置为RightToLeft，再把两个按钮拖到这个面板上。将TableLayoutPanel上面一行的大小设置为100%，调整下面一行的大小，使两个按钮刚好放下。现在你的编辑器就能平滑地调整大小了！

**1 建立一个简单的窗体。**

只需要一个文本框和两个按钮。另外把OpenFileDialog和SaveFileDialog控件也拖到窗体上。双击两个按钮，创建它们的事件处理程序，并为窗体增加一个名为name的私有string字段。不要忘记在最上面放上一个using System.IO语句。




**2 将Open按钮关联到openFileDialog。**

Open按钮显示一个OpenFileDialog，然后使用File.ReadAllText()将文件读入文本框：

```
private void open_Click(object sender, EventArgs e) {
  if (openFileDialog1.ShowDialog() == DialogResult.OK) {
    name = openFileDialog1.FileName;
    textBox1.Clear();
    textBox1.Text = File.ReadAllText(name);
  }
}
```

点击Open按钮会显示OpenFileDialog控件。

**3 下面关联Save按钮。**

Save按钮使用File.WriteAllText()方法保存文件：

```
private void save_Click(object sender, EventArgs e) {
  if (saveFileDialog1.ShowDialog() == DialogResult.OK) {
    name = saveFileDialog1.FileName;
    File.WriteAllText(name, textBox1.Text);
  }
}
```

ReadAllText()和WriteAllText()方法都是File类的方法。下一页就会介绍有关内容。后面几页将详细讨论这些方法。

**4 设置对话框的其他属性。**

★ 使用saveFileDialog的Title属性改变标题栏中的文本。

★ 设置initialFolder属性，让OpenFileDialog先显示一个指定的目录。

★ 使用Filter属性设置OpenFileDialog的过滤器，使它只显示文本文件。

如果没有增加过滤器，打开和保存对话框最下面的下拉列表将为空。可以尝试使用以下过滤器："Text Files (*.txt)|*.txt"

## Sharpen your pencil Solution

.NET有两个内置类，提供了大量静态方法来处理文件和文件夹。File类提供了处理文件的方法，另外可以使用Directory类处理目录。你认为以下各行代码分别有什么作用，请写下来。

| 代码 | 代码的作用 |
|---|---|
| `if (!Directory.Exists(@”c:\SYP”)) {`<br>`    Directory.CreateDirectory(@”c:\SYP”);`<br>`}` | 检查C:\SYP文件夹是否存在。如果不存在，则创建这个文件夹。 |
| `if (Directory.Exists(@”c:\SYP\Bonk”)) {`<br>`    Directory.Delete(@”c:\SYP\Bonk”);`<br>`}` | 检查C:\SYP\Bonk文件夹是否存在。如果存在这个文件夹，则将其删除。 |
| `Directory.CreateDirectory(@”c:\SYP\Bonk”);` | 创建目录C:\SYP\Bonk。 |
| `Directory.SetCreationTime(@”c:\SYP\Bonk”,`<br>`        new DateTime(1976, 09, 25));` | 将C:\SYP\Bonk文件夹的创建时间设置为1976年9月25日。 |
| `string[] files = Directory.GetFiles(@”c:\windows\”,`<br>`        ”*.log”, SearchOption.AllDirectories);` | 得到C:\Windows中与模式*.log匹配的所有文件的列表，这包括所有子目录中的所有匹配文件。 |
| `File.WriteAllText(@”c:\SYP\Bonk\weirdo.txt”,`<br>`        @”This is the first line`<br>`and this is the second line`<br>`and this is the last line”);` | 在C:\SYP\Bonk文件夹中创建一个名为“weirdo.txt”的文件（如果这个文件还不存在），并向其中写3行文本。 |
| `File.Encrypt(@”c:\SYP\Bonk\weirdo.txt”);`<br>这是CryptoStream的一种候选方法。 | 利用Windows的加密功能来加密文件“weirdo.txt”要使用登录帐户的凭证。 |
| `File.Copy(@”c:\SYP\Bonk\weirdo.txt”,`<br>`        @”c:\SYP\copy.txt”);` | 将C:\SYP\Bonk\weirdo.txt文件复制到C:\SYP\Copy.txt。 |
| `DateTime myTime =`<br>`        Directory.GetCreationTime(@”c:\SYP\Bonk”);` | 声明myTime变量，并设置为C:\SYP\Bonk文件夹的创建时间。 |
| `File.SetLastWriteTime(@”c:\SYP\copy.txt”, myTime);` | 修改C:\SYP\中copy.txt文件的最后写时间，使之等于myTime变量中存储的时间。 |
| `File.Delete(@”c:\SYP\Bonk\weirdo.txt”);` | 删除C:\SYP\Bonk\weirdo.txt文件。 |

# IDisposable确保对象适当地撤销

很多.NET类都实现了一个特别有用的接口，名为`IDisposable`。它只有一个成员：`Dispose()`方法。如果一个类实现了`IDisposable`，说明它撤销前需要做一些重要的事情，这往往是因为向它分配了一些资源，而且除非告诉它释放这些资源，否则资源不会收回。要利用`Dispose()`方法来告诉对象释放为它分配的资源。

**在一个using块中声明一个对象，这个对象的Dispose()方法会自动调用。**

可以使用IDE的"Go To Definition"特性找到C#的`IDispoable`官方定义。打开工程，在代码中的任何位置键入`IDisposable`。然后右键点击，并从菜单选择"Go To Definition"。它会打开一个新的标签页，其中包含IDisposable的代码。展开所有代码，你会看到：

后面还会介绍"Go To Definition"。

```
namespace System
{
    // Summary:
    //     Defines a method to release allocated resources.
    public interface IDisposable
    {
        // Summary:
        //     Performs application-defined tasks
        //     associated with freeing, releasing, or
        //     resetting unmanaged resources.
        void Dispose();
    }
}
```

很多类会分配重要的资源，如内存、文件和其他对象。这说明类会把这些资源拿走，在你告诉它已经用完这些资源之前是不会归还的。

对于任何实现了IDisposable接口的类，只要调用了它的Dispose()方法，就会立即释放所占用的资源。这往往是结束对象处理的最后一步。

**Go To Definition**

这是IDE中的一个很方便的特性，利用它能自动跳至任何变量、对象或方法的定义。只需右键点击变量或方法，并选择"Go To Definition"，IDE就会自动跳至它的定义代码。也可以不使用菜单，按下F12也能达到这个目的。

**al-lo-cate，动词.**

为完成一个特定目的分配资源或职责。编程团队对他们的项目经理很恼火，因为他把所有会议室都分配给一个毫无意义的管理层研讨会。

# 利用using语句避免文件系统错误

这一章一直在告诉你需要关闭所使用的流。这是因为，程序员最常遇到的bug就是因为他们处理文件时没有适当关闭流造成的。幸运的是，C#提供了一个很棒的工具，可以确保这一切不会在你身上发生，这个工具就是IDisposable和Dispose()方法。将使用流的代码包围在一个using语句中时，它会自动为你关闭流。你要做的只是用一个using语句来声明流引用，后面是使用了这个引用的代码块（放在大括号里）。这样一来，只要运行完这个代码块，using语句会自动地调用流的Dispose()方法。具体工作如下：

这些"using"语句与代码最上面的using代码行不同。

using语句后面总有一个对象声明……

……然后是一个用大括号括起的代码块。

```
using (StreamWriter sw = new StreamWriter("secret _ plan.txt")) {
    sw.WriteLine("How I'll defeat Captain Amazing");
    sw.WriteLine("Another genius secret plan");
    sw.WriteLine("by The Swindler");
}
```

这些语句可以正常地使用using语句中创建的对象（与使用普通对象并无不同）。

using语句结束时，会运行所用对象的Dispose()方法。

在这种情况下，所用对象由sw指示（在using语句中声明），所以Stream类的Dispose()方法会运行……这就会关闭流。

**每个流都有一个Dispose()方法，它会关闭这个流。所以如果在一个using语句中声明流，它就会自行关闭！**

## 多个对象要使用多个using语句

可以在using语句之上再罗列其他using语句，不必另外使用大括号或缩进。

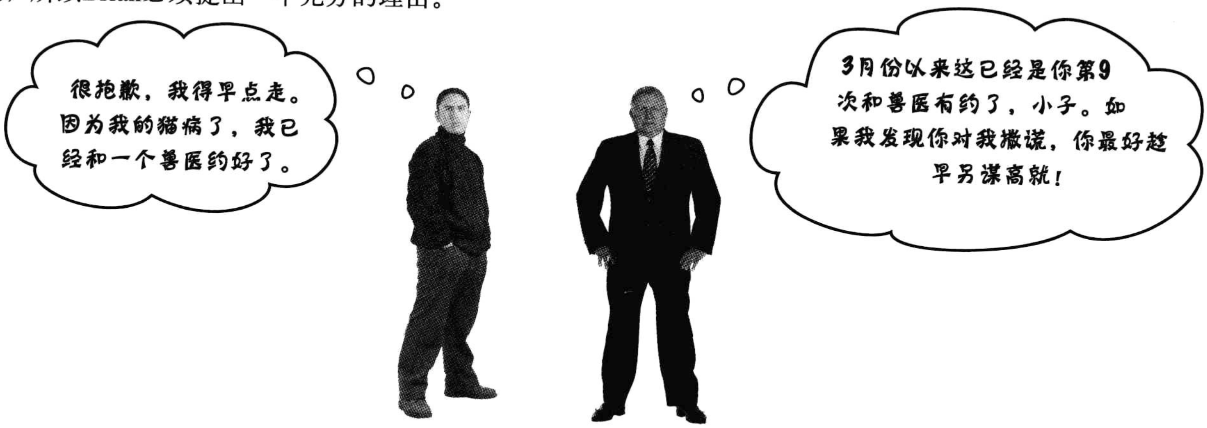
```
using (StreamReader reader = new StreamReader("secret _ plan.txt"))
using (StreamWriter writer = new StreamReader("email.txt"))
{
    // statements that use reader and writer
}
```

现在不需要对流调用Close()了，因为using语句会自动关闭流。

**使用流时，一定要在一个using语句中声明。这样能确保流总会关闭！**

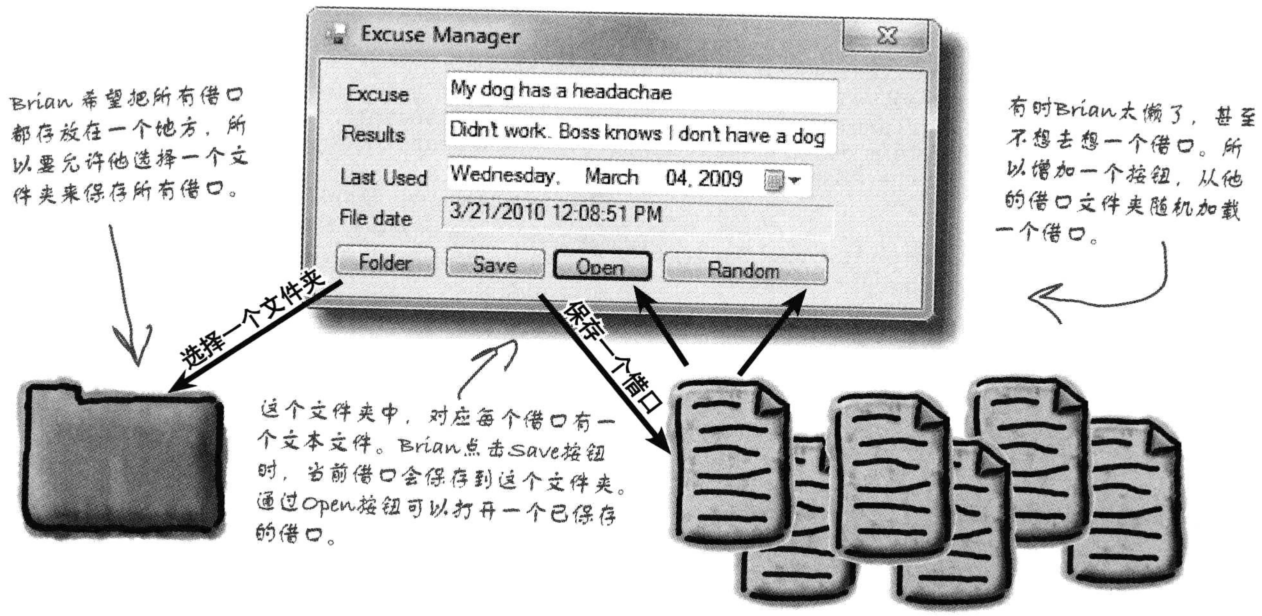
# 工作上的麻烦

假设你遇到了Brian。他是一个C#开发人员，很热爱自己的工作，不过他喜欢时不时地请假。但是他的老板很不喜欢下属请假，所以Brian必须提出一个充分的理由。




## 可以建立一个程序帮Brian管理他的借口

使用读写文件的有关知识来建立一个借口管理系统，Brian可以使用这个系统跟踪最近用过哪些借口，以及这些借口对老板的效果如何。



**Exercise**

构建借口管理系统，使Brian能够管理他的请假借口。

| Excuse |
|---|
| Description: string<br>Results: string<br>LastUsed: DateTime<br>ExcusePath: string |
| OpenFile(string)<br>Save(string) |

**1** **创建窗体。**

这个窗体有一些特殊特性：

- ★ 第一次加载窗体时，只有Folder按钮启用，用户选择一个文件夹之前，所有其他三个按钮都要禁用。
- ★ 窗体打开或保存一个借口时，它会使用一个Label控件显示借口文件的文件日期，这个控件的AutoSize设置为False，BorderStyle设置为Fixed3D。
- ★ 保存一个借口后，窗体会弹出一个“Excuse Written”消息框。
- ★ Folder按钮弹出一个文件夹浏览对话框。如果用户选择了一个文件夹，将启用Save、Open和Random Excuse按钮。
- ★ 窗体知道何时有尚未保存的修改。如果没有未保存的修改内容，窗体标题栏上的文本是“Excuse Manager”。不过，如果用户修改了3个域中的任何一个域，窗体会在标题栏上增加一个星号（*）。数据保存后或者打开一个新借口时这个星号会消失。
- ★ 窗体需要跟踪当前文件夹以及当前借口是否已保存。可以使用3个输入控件的Changed事件处理程序来确定借口是否保存。

将一个文本框拖到窗体并双击文本框时，就会为这个域创建一个Changed事件处理程序。

**2** **创建一个Excuse类，并在窗体中存储它的一个实例。**

现在向窗体增加一个CurrentExcuse字段来保存当前借口。需要3个重载构造函数：一个在第一次加载窗体时使用，另一个用于打开一个文件，还有一个用于打开一个随机的借口。增加OpenFile()方法来打开借口（将在构造函数中使用这个方法），另外增加Save()方法来保存借口。然后增加以下UpdateForm()方法更新控件（由这个方法，应该能得到关于这个类的一些提示）：

```
private void UpdateForm(bool changed) {
  if (!changed)  {
     this.description.Text = currentExcuse.Description;
     this.results.Text = currentExcuse.Results;
     this.lastUsed.Value = currentExcuse.LastUsed;
     if (!String.IsNullOrEmpty(currentExcuse.ExcusePath))
       FileDate.Text = File.GetLastWriteTime(currentExcuse.ExcusePath).ToString();
       this.Text = "Excuse Manager";
   }
   else
       this.Text = "Excuse Manager*";
   this.formChanged = changed;
}
```

这个参数指示窗体是否改变。窗体中需要有一个字段跟踪这个状态。

要记住，!表示非（NOT），所以这会检查借口路径是否非null或非空。

双击输入控件，使IDE为你建立Changed事件处理程序。3个输入控件的事件处理程序首先会修改Excuse实例，然后调用UpdateForm(true)，再由你来改变窗体上的域。

确保在窗体的构造函数中初始化借口的LastUsed值。

```
public Form1() {
    InitializeComponent();
    currentExcuse.LastUsed = lastUsed.Value;
}
```

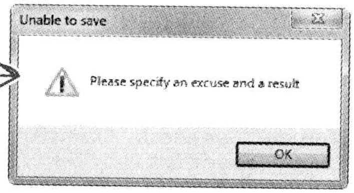
**3** **由Folder按钮打开一个文件夹浏览器。**

用户点击Folder按钮时，窗体要弹出一个“Browse for Folder”对话框。这个窗体需要用一个字段存储这个文件夹，以便其他对话框使用。窗体第一次加载时，Save、Open和Random Excuse按钮都是禁用的，但是如果用户选择了一个文件夹，Folder按钮就会启用这些按钮。

**4** **由Save按钮将当前借口保存到一个文件。**
点击Save按钮应当显示一个另存为（Save As）对话框。

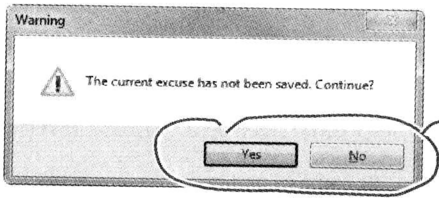
- ★ 每个借口都保存到一个单独的文本文件。文件的第一行是借口，第二行是结果，第三行是最近使用的日期（使用DateTimePicker的ToString()方法）。Excuse类应当有一个Save()方法将借口保存到一个指定的文件。
- ★ 打开另存为对话框时，文件夹应当设置为用户使用Folder按钮选择的文件夹，filename应当设置为借口再加一个“.txt”扩展名。
- ★ 这个对话框应当有两个过滤器：Text Files (*.txt)和All Files (*.*)。
- ★ 如果用户试图保存当前借口，但是借口或结果为空，窗体会弹出一个警告对话框：

使用允许指定一个MessageBoxIcon参数的重载MessageBox.Show()方法，可以显示这个感叹号。




**5** **由Open按钮打开一个已保存的借口。**
点击Open按钮应当弹出一个打开（Open）对话框。

- ★ 打开Open对话框时，其文件夹应当设置为用户使用Folder按钮选择的文件夹。
- ★ 向Excuse类增加一个Open()方法，从一个指定文件打开借口。
- ★ 使用Convert.ToDateTime()将保存的日期加载到DateTimePicker控件。
- ★ 如果用户试图打开当前借口，但是当前借口未保存，要弹出以下对话框：




使用允许指定MessageBoxButtons.YesNo参数的重载MessageBox.Show()方法，可以显示一个Yes/No对话框。如果用户点击“No”，Show()返回DialogResult.No。

**6** **最后，由Random Excuse按钮加载一个随机的借口。**
用户点击Random Excuse按钮时，会在借口文件夹中查找文件，随机地选择其中一个借口并打开。

- ★ 窗体需要在一个字段中保存一个Random对象，并把它传递到Excuse对象的某个重载构造函数。
- ★ 如果当前借口未保存，与Open按钮一样，这个按钮也会弹出同样的警告对话框。

**Exercise Solution**

构建借口管理系统，使Brian能够管理他的请假借口。

```
private Excuse currentExcuse = new Excuse();
private string selectedFolder = "";
private bool formChanged = false;
Random random = new Random();
```

窗体使用一些字段来存储当前Excuse对象、所选文件夹，并记住当前借口是否改变，另外还要为Random Excuse按钮保存一个Random对象。

```
private void folder_Click(object sender, EventArgs e) {
    folderBrowserDialog1.SelectedPath = selectedFolder;
    DialogResult result = folderBrowserDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        selectedFolder = folderBrowserDialog1.SelectedPath;
        save.Enabled = true;
        open.Enabled = true;
        randomExcuse.Enabled = true;
    }
}
```

如果用户选择了一个文件夹，窗体将保存文件夹名，然后启用另外3个按钮。

两个竖线表示或（OR），如果description为空或者results为空，这就为true。

```
private void save_Click(object sender, EventArgs e) {
    if (String.IsNullOrEmpty(description.Text) || String.IsNullOrEmpty(results.Text)) {
        MessageBox.Show("Please specify an excuse and a result",
             "Unable to save", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        return;
    }
    saveFileDialog1.InitialDirectory = selectedFolder;
    saveFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
    saveFileDialog1.FileName = description.Text + ".txt";
    DialogResult result = saveFileDialog1.ShowDialog();
    if (result == DialogResult.OK) {
        currentExcuse.Save(saveFileDialog1.FileName);
        UpdateForm(false);
        MessageBox.Show("Excuse written");
    }
}
```

在这里为“另存为”对话框设置过滤器。

这会导致保存对话框最下面“Files of Type”下拉列表中显示两行：一行是Text Files (*.txt)，另一行是All Files (*.*)。

```
private void open_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        openFileDialog1.InitialDirectory = selectedFolder;
        openFileDialog1.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
        openFileDialog1.FileName = description.Text + ".txt";
        DialogResult result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK) {
            currentExcuse = new Excuse(openFileDialog1.FileName);
            UpdateForm(false);
        }
    }
}
```

使用打开和保存对话框返回的DialogResult enum，确保只有当用户点击“OK”时（而不是“Cancel”）才打开或保存。

```
private void randomExcuse_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

```
private bool CheckChanged() {
    if (formChanged) {
        DialogResult result = MessageBox.Show(
              "The current excuse has not been saved. Continue?",
               "Warning", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
        if (result == DialogResult.No)
            return false;
    }
    return true;
}

private void description_TextChanged(object sender, EventArgs e) {
    currentExcuse.Description = description.Text;
    UpdateForm(true);
}

private void results_TextChanged(object sender, EventArgs e) {
    currentExcuse.Results = results.Text;
    UpdateForm(true);
}

private void lastUsed_ValueChanged(object sender, EventArgs e) {
    currentExcuse.LastUsed = lastUsed.Value;
    UpdateForm(true);
}

class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }
    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.txt");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        using (StreamReader reader = new StreamReader(excusePath)) {
            Description = reader.ReadLine();
            Results = reader.ReadLine();
            LastUsed = Convert.ToDateTime(reader.ReadLine());
        }
    }
    public void Save(string fileName) {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            writer.WriteLine(Description);
            writer.WriteLine(Results);
            writer.WriteLine(LastUsed);
        }
    }
}
```

MessageBox.Show()也返回一个DialogResult enum可供检查。

对应窗体上的3个输入域，有3个Changed事件处理程序。如果触发其中任意一个事件处理程序，说明借口有改变，所以首先更新Excuse实例，然后调用UpdateForm()，向窗体的标题栏增加星号，并设置Changed为true。

向UpdateForm()传入true，告诉它只是将窗体标志为已经改变，但不要更新输入控件。

Random Excuse按钮使用了Directory.GetFiles()方法将所选文件夹中的所有文本文件读入一个数组，然后随机选择一个数组索引打开相应文件。

确保每次打开一个流时都使用using语句。这样一来，就能保证文件总会关闭。

在这里使用using语句。在一个using语句中声明StreamWriter，这样就会自动调用其Close()方法。

调用了LastUsed.ToString()吗？要记住，WriteLine()会自动调用这个方法！

# 写文件通常要做大量决策

你可能会写很多这样的程序：它们取一个输入（可能来自一个文件），必须根据这个输入来确定要做什么。下面的代码使用了一个长长的if语句，这很典型。它要检查part变量，并根据它使用的enum值向文件打印不同的文本行。这里有很多选择，所以也有很多else if：

```
enum BodyPart {
    Head,
    Shoulders,
    Knees,
    Toes
}
```

这是一个enum，我们希望将一个变量与4个枚举成员分别比较，并根据匹配结果向StreamWriter写不同的文本行。如果它们都不匹配，则写一个不同的内容。

```
private void WritePartInfo(BodyPart part, StreamWriter writer) {
  if (part == BodyPart.Head)
    writer.WriteLine("the head is hairy");
  else if (part == BodyPart.Shoulders)
    writer.WriteLine("the shoulders are broad");
  else if (part == BodyPart.Knees)
    writer.WriteLine("the knees are knobby");
  else if (part == BodyPart.Toes)
    writer.WriteLine("the toes are teeny");
  else
    writer.WriteLine("some unknown part is unknown");
}
```

如果使用一系列if/else语句，就会反复地写这个"if (part ==[option])"。

这里加了最后的else，以备无法找到任何匹配。

**BRAIN POWER**

写代码时如果有这么多if/else语句，哪些方面可能出问题？请考虑括号、单个等号（而不是==比较操作符）等可能导致的键入错误和bug。

# 使用switch语句做出正确的选择

将一个变量与一组不同的值做比较，这是你将反复看到的一个常见模式。读写文件时这种操作尤其常见。实际上，正是因为它如此常见，所以C#专门为此设计了一种特殊的语句。

switch语句与文件并没有特别的关联。这只是一个有用的C#工具，可以在这里使用。

利用switch语句，可以将一个变量与多个值比较，而且代码更易读，也很紧凑。以下switch语句的作用与上一页一系列if/else语句作用完全相同：

**switch语句将一个变量与多个可能值进行比较。**

```
enum BodyPart
{
    Head,
    Shoulders,
    Knees,
    Toes,
}

private void WritePartInfo(BodyPart part, StreamWriter writer)
{
    switch (part) {
      case BodyPart.Head:
        writer.WriteLine("the head is hairy");
        break;
      case BodyPart.Shoulders:
        writer.WriteLine("the shoulders are broad");
        break;
      case BodyPart.Knees:
        writer.WriteLine("the knees are knobby");
        break;
      case BodyPart.Toes:
        writer.WriteLine("the toes are teeny");
        break;
      default:
        writer.WriteLine("some unknown part is unknown");
        break;
    }
}
```

首先是switch关键字，后面是要与一组候选值比较的变量。

switch语句的体是一系列case语句，将switch关键字后面的变量与一个特定值比较。

每个case都必须以"break;"结束，这样C#才能知道一种情况在哪里结束，下一个在从哪里开始。

还可以用"return"结束一个case，只要一个case不会"落入"下一个case，程序就能编译。

每个case都包括一个case关键字，后面是要比较的值和一个冒号。然后是一系列语句，最后是"break;"。如果这个case中的值与比较值匹配则执行这些语句。

switch语句最后必须是一个default:块，如果所有其他case都不匹配，就要执行这个代码块。

# 使用switch语句从文件读写一副牌

将一副牌写到文件中很容易，只需建立一个循环，把各个牌名写入一个文件。可以向Deck对象增加下面这个方法完成这个工作：

```
public void WriteCards(string filename) {
    using (StreamWriter writer = new StreamWriter(filename)) {
        for (int i = 0; i < cards.Count; i++) {
            writer.WriteLine(cards[i].Name);
        }
    }
}
```

但是怎么读文件呢？那可没有这么容易。现在可以用到switch语句了。

**利用switch语句，可以测试一个值，将它与一组case中的值比较，并根据其匹配情况执行不同的语句。**

```
Suits suit;
switch (suitString) {
    case "Spades":
        suit = Suits.Spades;
        break;
    case "Clubs":
        suit = Suits.Clubs;
        break;
    case "Hearts":
        suit = Suits.Hearts;
        break;
    case "Diamonds":
        suit = Suits.Diamonds;
        break;
    default:
        MessageBox.Show(suitString + " isn't a valid suit!");
}
```

switch语句首先有一个要比较的值。这个switch语句在一个方法中调用，该方法在一个string变量中（suitString）存储了花色。

每个case语句将一个值与switch行指定的值进行比较。如果二者匹配，则执行后面的所有语句，直到遇到一个break。

default放在最后。如果以上所有case都不匹配，就会执行default后面的语句。

# 增加一个重载Deck()构造函数，从文件读入一副牌

可以使用switch语句为上一章编写的Deck类建立一个新的构造函数。这个构造函数读入一个文件，检查有关扑克牌的每一行。只要扑克牌合法就增加到这副牌中。

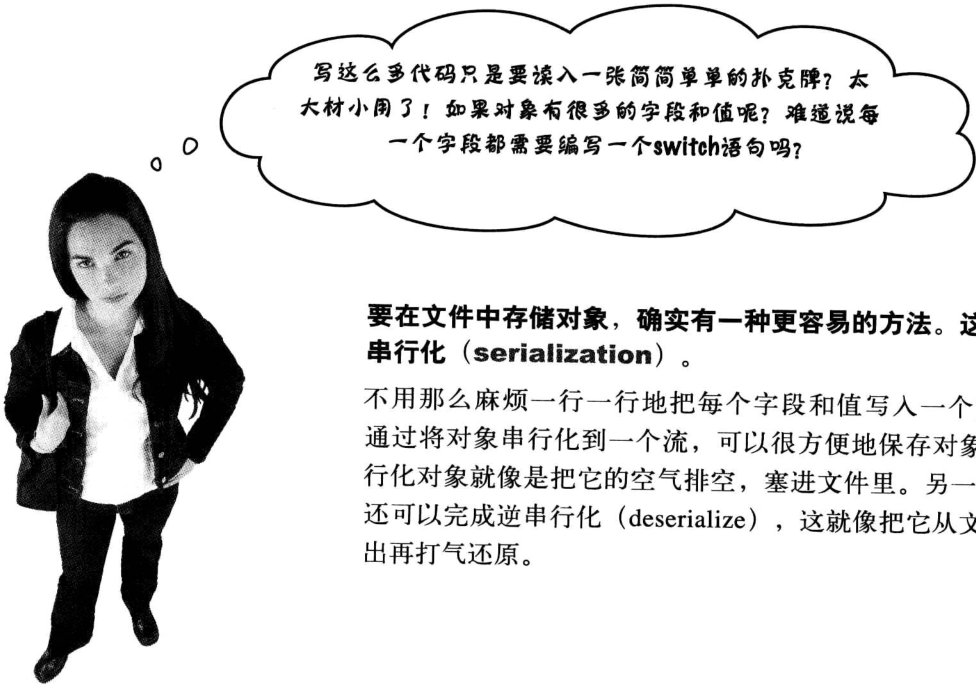
所有string都有一个`Split()`方法，这个方法很有用。利用这个方法，可以把串分解为一个子串数组，为此只需传入一个分隔符数组`char[]`，将利用这些分隔符来分解串。

```
public Deck(string filename) {
    cards = new List<Card>();
    StreamReader reader = new StreamReader(filename);
    while (!reader.EndOfStream) {
        bool invalidCard = false;
        string nextCard = reader.ReadLine();
        string[] cardParts = nextCard.Split(new char[] { ' ' });
        Values value = Values.Ace;
        switch (cardParts[0]) {
            case "Ace": value = Values.Ace; break;
            case "Two": value = Values.Two; break;
            case "Three": value = Values.Three; break;
            case "Four": value = Values.Four; break;
            case "Five": value = Values.Five; break;
            case "Six": value = Values.Six; break;
            case "Seven": value = Values.Seven; break;
            case "Eight": value = Values.Eight; break;
            case "Nine": value = Values.Nine; break;
            case "Ten": value = Values.Ten; break;
            case "Jack": value = Values.Jack; break;
            case "Queen": value = Values.Queen; break;
            case "King": value = Values.King; break;
            default: invalidCard = true; break;
        }
        Suits suit = Suits.Clubs;
        switch (cardParts[2]) {
            case "Spades": suit = Suits.Spades; break;
            case "Clubs": suit = Suits.Clubs; break;
            case "Hearts": suit = Suits.Hearts; break;
            case "Diamonds": suit = Suits.Diamonds; break;
            default: invalidCard = true; break;
        }
        if (!invalidCard) {
            cards.Add(new Card(suit, value));
        }
    }
}
```

这行代码告诉C#使用空格作为分隔符来分解nextCard串。这会把串“Six of Diamonds”分解为数组{“Six”，“of”，“Diamonds”}。

这个switch语句检查文本行中的第一个词，查看它是否与某个牌面大小匹配。如果匹配，则为value变量指定适当的值。

对文本行中的第3个词做同样的处理，不过要用来检查花色。




**要在文件中存储对象，确实有一种更容易的方法。这称为串行化（serialization）。**

不用那么麻烦一行一行地把每个字段和值写入一个文件，通过将对象串行化到一个流，可以很方便地保存对象。串行化对象就像是把它的空气排空，塞进文件里。另一方面，还可以完成逆串行化（deserialize），这就像把它从文件取出再打气还原。

Ok。这里需要说明一下：还有一个名为Enum.Parse()的方法，有关内容将在第14章介绍，它会把串“Spades”转换为enum值Suits.Spades。不过，在这里串行化更有意义。稍后你就会了解……

## 对象串行化时发生了什么？

把对象从堆中取出并复制到一个文件中，这看起来很神秘，对象肯定发生了一些不同寻常的事情，但实际上这确实很简单。

**❶ 堆中的对象。**

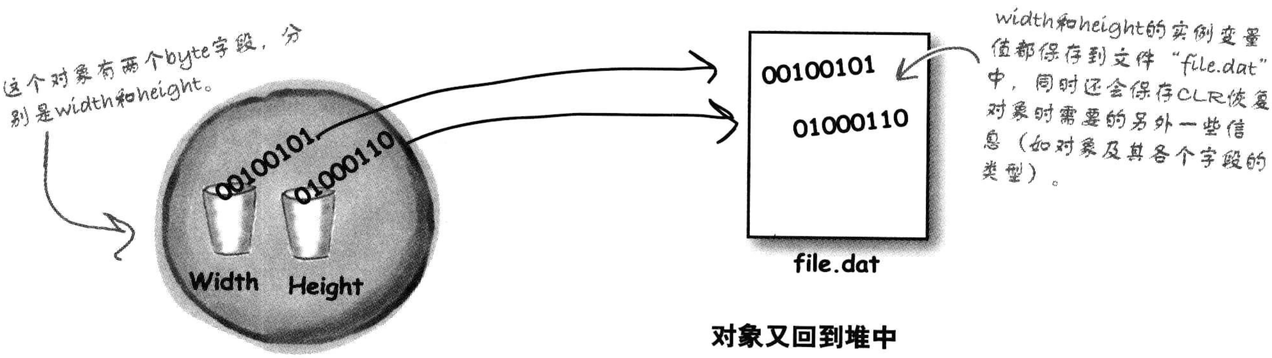


创建一个对象的实例时，它有一个状态（state）。正是因为对象特有的状态，才使得一个类的一个实例与同一个类的另一个实例有所不同。

**❷ 串行化的对象。**



C#串行化一个对象时，它会保存这个对象的完整状态，从而就能让一个相同的实例（对象）在堆中复活。




**对象又回到堆中**



**❸ 然后……**

后来，可能是几天以后，而且在另一个程序中—可以再对这个文件逆串行化。这会把原来的类从文件中取出来，并恢复成原先的样子，它的所有字段和值都原封不动地保留。

# 到底什么是对象的状态？哪些需要保存？

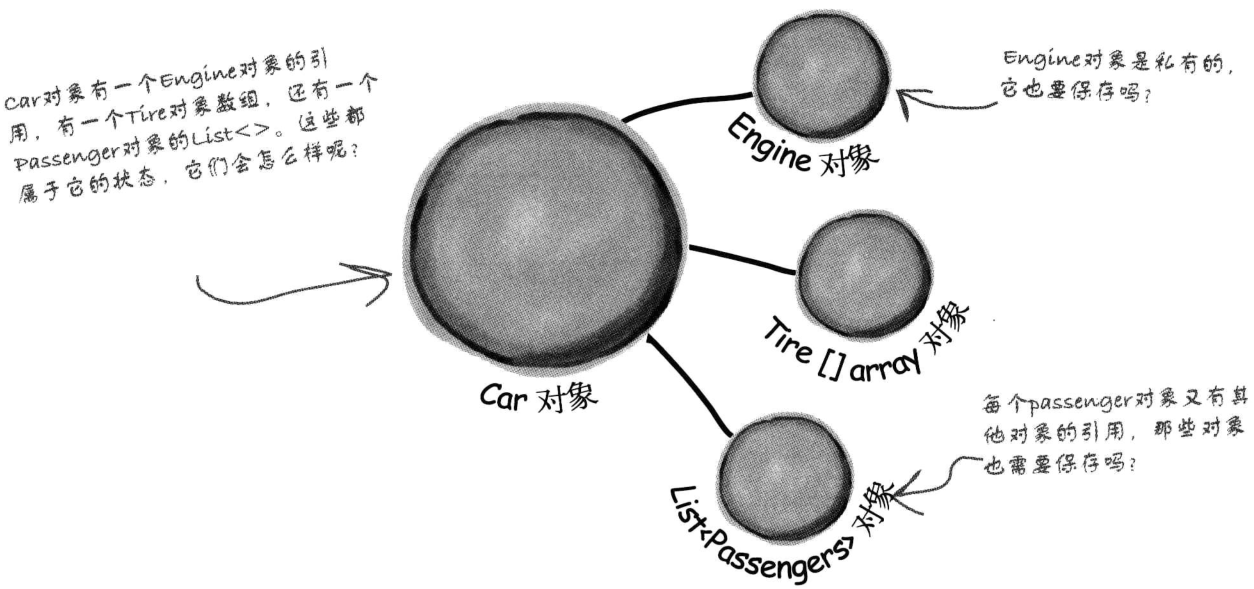
我们已经知道，对象会在字段中存储它的状态。所以串行化一个对象时，所有字段都必须保存到文件中。

对象比较复杂时，串行化才更有意思。37和70是byte类型，这些都是值类型，所以可以原样地写到文件中。但是如果一个对象有一个实例变量，而这个实例变量是一个对象引用，又会怎么处理呢？如果一个对象有5个实例变量，它们都是对象引用，会怎么样？如果这些对象实例变量本身又有实例变量呢？

好好想一想。对象的哪些部分可能是独一无二的？想象一下，要得到一个与所保存的对象完全相等的对象，为此需要恢复哪些内容。要以某种方式把堆中的所有一切都写至文件。

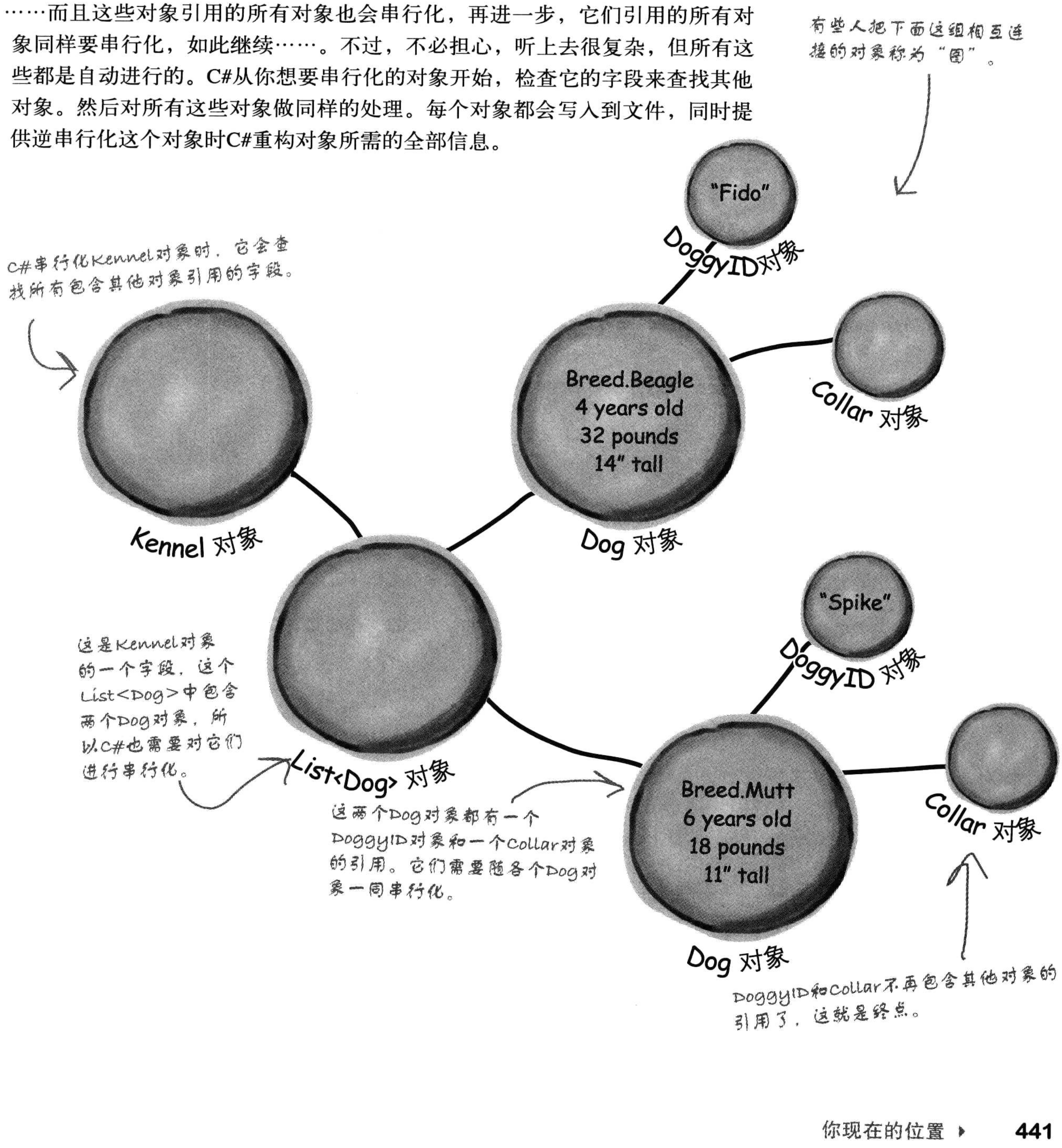
## BRAIN BARBELL

保存这个Car对象，以便以后恢复为原来的状态，为此要做些什么？假设这辆车上有3个乘客，一个3.0排量的发动机，还有全天候子午线轮胎，这些不都是这个Car对象的状态吗？它们会怎么样呢？



# 对象串行化时，它引用的所有对象也要串行化……

……而且这些对象引用的所有对象也会串行化，再进一步，它们引用的所有对象同样要串行化，如此继续……。不过，不必担心，听上去很复杂，但所有这些都是自动进行的。C#从你想要串行化的对象开始，检查它的字段来查找其他对象。然后对所有这些对象做同样的处理。每个对象都会写入到文件，同时提供逆串行化这个对象时C#重构对象所需的全部信息。



# 利用串行化可以一次读写整个对象

**将对象复制到文件或从文件读出一个对象很快捷。可以完成串行化和逆串行化。**

并不仅限于向文件读写文本行。可以使用串行化让程序将整个对象复制到文件，还可以将其读出…… 所有这些只需要几行代码就能做到！在此之前需要先做一些准备工作，在要串行化的类最前面增加一行[Serializable]，不过仅此而已，一旦增加了这行代码，就一切准备就绪，可以写文件了。

## 需要一个BinaryFormatter对象

如果想串行化一个对象（可以是任何对象），首先要创建一个BinaryFormatter实例。这很简单，只需要一行代码（当然，还要在类文件最上面增加一行using代码）。

```
using System.Runtime.Serialization.Formatters.Binary;
:.:.:
BinaryFormatter formatter = new BinaryFormatter();
```

## 现在只需创建一个流，然后读写对象

使用BinaryFormatter对象的Serialize()方法可以将任何对象写到一个流中。

File.Create()方法创建一个新文件。可以使用File.OpenWrite()打开一个已有的文件。

```
using (Stream output = File.Create(filenameString)) {
    formatter.Serialize(output, objectToSerialize);
}
```

Serialize()方法取一个对象，并把它写至一个流。这比自己编写一个方法把对象写到文件中要容易得多。

一旦将对象串行化到一个文件，还可以使用BinaryFormatter对象的Deserialize()方法再把对象读回。这个方法返回一个引用，所以要把它复制到一个引用变量时，需要先对输出进行强制类型转换，使它与这个引用变量的类型匹配。

```
using (Stream input = File.OpenRead(filenameString)) {
    SomeObj obj = (SomeObj)formatter.Deserialize(input);
}
```

使用Deserialize()将对象从流读回时，不要忘记对返回值进行强制类型转换，使它与所读的对象类型匹配。

# 如果希望类可以串行化，要用[Serializable]属性来标志

属性（attribute）是一种可以增加到任何C#类前面的特殊标记。C#通过这些属性来存储有关代码的元数据，或如何使用或处理代码的有关信息。在一个类前面增加[Serializable]时（放在类声明前面），就是在告诉C#：这个类可以安全地串行化。另外，只有当类中包含的字段是值类型（如int、string或enum），或者是其他可串行化的类时，才能用[Serializable]属性标志这个类。如果没有向需要串行化的类增加这个属性，或者如果类中某个字段的类型不能串行化，运行程序时就会得到一个错误。你可以自己试试看……

**动手做**

1. **创建一个类并串行化。**

还记得第3章的Guy类吗？下面串行化Joe，这样就能保留一个文件，即使关闭了程序也能知道Joe口袋里有多少钱。

```
[Serializable]
 class Guy
```

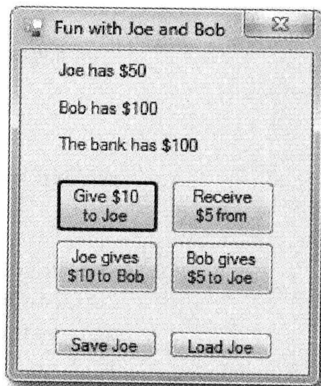
需要在类的最前面增加这个属性，才能对它串行化。

以下代码把这个对象串行化到一个名为“Guy_file.dat”的文件，另外向窗体增加一个“Save Joe”按钮和一个“Load Joe”按钮。

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...

private void saveJoe_Click(object sender, EventArgs e)
{
  using (Stream output = File.Create("Guy_File.dat")) {
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(output, joe);
  }
}
private void loadJoe_Click(object sender, EventArgs e)
{
  using (Stream input = File.OpenRead("Guy_File.dat")) {
    BinaryFormatter formatter = new BinaryFormatter();
    joe = (Guy)formatter.Deserialize(input);
  }
  UpdateForm();
}
```

必须有这两行using代码。第一个对应文件和流方法，第二个对应串行化。




2. **运行程序，体验一下。**

假设在运行程序期间Joe通过与Bob的交易存了200元钱，如果只是因为你要退出程序就让他失去所有这些钱，他肯定不高兴。现在你的程序可以把Joe保存到一个文件，需要时还可以把他恢复回来。

# 对一副牌串行化和逆串行化

取一副牌，把它写到一个文件。C#中串行化对象非常容易。只需要创建一个流，然后写对象。

动手做

**❶ 创建一个新工程，增加Deck和Card类。**

在Solution Explorer中右键点击这个工程，并选择“Add/Existing Item”，增加第8章Go Fish!游戏中创建的Card和Deck类（以及Suits和Values enums，还有CardComparer_bySuit和CardComparer_byValue接口）。还需要增加两个扑克牌比较类，因为Deck要用到这两个比较类。IDE会把这些文件复制到新工程，别忘了修改每个类文件最前面的namespace行，要与新工程的命名空间一致。

**❷ 将所有类标志为可串行化。**

向增加到工程的所有类增加[Serializable]属性。

如果没有这样做，C#不允许将类串行化到文件。

**❸ 向窗体增加两个有用的方法。**

RandomDeck方法创建随机的一副牌，DealCards方法将所有牌出光，并显示到控制台。

这会创建一副空牌，然后使用上一章的Card类随机增加一些牌。

```
Random random = new Random();
private Deck RandomDeck(int number) {
    Deck myDeck = new Deck(new Card[] { });
    for (int i = 0; i < number; i++)
    {
        myDeck.Add(new Card(
            (Suits)random.Next(4),
            (Values)random.Next(1, 14)));
    }
    return myDeck;
}

private void DealCards(Deck deckToDeal, string title) {
    Console.WriteLine(title);
    while (deckToDeal.Count > 0)
    {
        Card nextCard = deckToDeal.Deal(0);
        Console.WriteLine(nextCard.Name);
    }
    Console.WriteLine("-----------------");
}
```

DealCards()方法出光这副牌中的每一张牌，并显示到控制台。

**4** **好了，已经准备就绪……下面串行化这副牌。**

首先增加一些按钮将一副随机的牌串行化到一个文件，再将其读回。查看控制台输出，确保写出的这副牌与读入的牌完全相同。

```
private void button1_Click(object sender, EventArgs e) {
    Deck deckToWrite = RandomDeck(5);
    using (Stream output = File.Create("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      bf.Serialize(output, deckToWrite);
    }
    DealCards(deckToWrite, "What I just wrote to the file");
}
```

BinaryFormatter对象可以取任何标志有Serializable属性的对象，在这里就是一个Deck对象，并使用Serialize()方法把对象写至一个流。

```
private void button2_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck1.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      Deck deckFromFile = (Deck)bf.Deserialize(input);
      DealCards(deckFromFile, "What I read from the file");
    }
}
```

BinaryFormatter的Deserialize()方法返回一个Object，这是所有C#对象都要继承的最一般的基类型，因此需要把它强制转换为一个Deck对象。

**5** **下面将多副牌串行化到同一个文件。**

一旦打开一个流，可以根据需要写很多内容。可以把多个对象串行化到同一个文件。所以，下面再增加两个按钮，把多个deck对象（个数随机）写入文件。查看输出确保一切正常。

```
private void button3_Click(object sender, EventArgs e) {
    using (Stream output = File.Create("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
        Deck deckToWrite = RandomDeck(random.Next(1,10));
        bf.Serialize(output, deckToWrite);
        DealCards(deckToWrite, "Deck #" + i + " written");
      }
    }
}
```

可以依次将对象串行化到同一个流。

要注意从文件读Deck对象的那行代码，这里使用了(Deck)将Deserialize()的输出强制转换为一个Deck。这是因为Deserialize()返回一个object，但是不一定知道这个对象是何种类型。

```
private void button4_Click(object sender, EventArgs e) {
    using (Stream input = File.OpenRead("Deck2.dat")) {
      BinaryFormatter bf = new BinaryFormatter();
      for (int i = 1; i <= 5; i++) {
          Deck deckToRead = (Deck)bf.Deserialize(input);
          DealCards(deckToRead, "Deck #" + i + " read");
      }
    }
}
```

只要求将从流读出的对象强制转换为正确的类型，至于能串行化多少个对象是没有限制的。

**6** **来看你写的文件。**

在记事本中打开Deck1.dat文件（File.Create()将这个文件创建在工程目录下的bin\Debug文件夹中）。里面的内容可能读不懂，不过这确实包含了恢复整副牌所需的全部信息。



等一下，把对象写到一些奇怪的文件里，而且打开这些文件时我看到的很像是一堆垃圾，我实在不明白这样做有什么意义。要知道，如果我把一副牌作为串写入文件，就能在记事本里打开输出，查看其中的所有内容。C#不是想让我更好地了解自己做的事情吗？

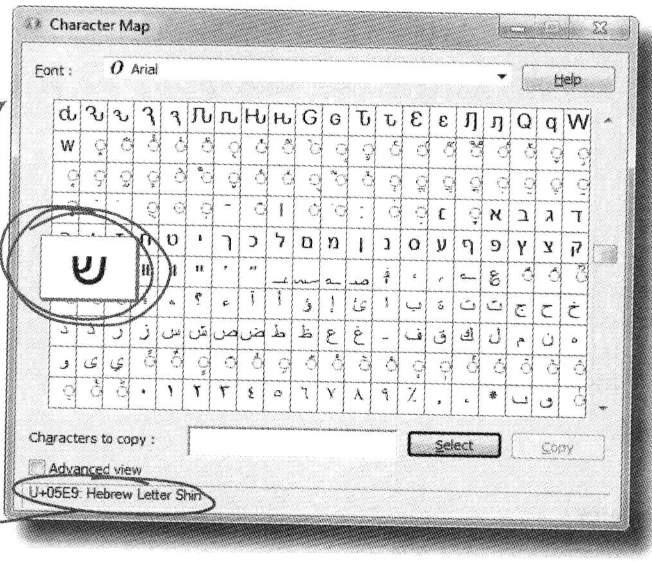
**将对象串行化到一个文件时，它们会以二进制格式写入文件。**

不过这并不意味着不可辨识，这样做只是为了更为紧凑。正是因为这一点，打开包含串行化对象的文件时，如果原对象中包含串，完全可以在文件中识别出来：因为C#将串写入文件时最紧凑的方式就是作为串写入。不过要把一个数字作为串写入就有些浪费了。存储int只需4个字节。所以如果C#把一个数（比如说49,369,144）存储为一个8字符的串（如果加上逗号就是10个字符）来方便你阅读，这就不正常了。这样会浪费大量空间！

## 在幕后

.NET使用Unicode格式将字符或串编码为字节。幸运的是，Windows有一些很有用的小工具，可以帮助我们了解unicode是如何工作的。打开字符映射表（Start菜单下选择 “Accessories”→“System Tools”可以看到字符映射表的菜单，或者选择“Start”→“Run”并键入“charmap.exe”就可以打开这个工具）。

查看世界上各种语言中使用的所有字母和符号时，你会发现，仅仅存储文本也有很大差别，需要向文件写入不同的内容。针对这个问题，.NET采用一种称为Unicode的格式对所有串和字符进行编码。编码（Encoding）是指将逻辑数据（如字母H）转换为字节（数字72）。需要这样做是因为字母、数字、枚举和其他数据最后都会以字节形式存储在磁盘或内存中。也因为这一点，字符映射表很有用，它能显示字母如何编码为数字。




选择Arial字体，向下滚动到希伯来字母。找到字母Shin，并点击这个字母。

一旦点击这个字母，就会在状态条上显示出它的Unicode码。希伯来字母Shin的编码是数字05E9。这是一个十六进制数。十六进制简称为“hex”。

可以使用Windows计算器把它转换为十进制数：打开计算器，选择科学模式，点击“Hex”单选钮，输入“05E9”，再点击“Dec”，就能得到1 513。

Unicode是非赢利组织Unicode协会开发的一个行业标准，它适用于所有程序以及不同的计算机平台。可以花点时间访问Unicode网站（http://www.unicode.org/）来加深了解。

# .NET使用Unicode存储字符和文本

C#用于存储文本的两个类型string和char，都采用Unicode格式在内存中存储数据。数据作为字节写至一个文件时，就是将其Unicode码写入文件。下面创建一个新工程，向窗体拖入3个按钮，我们将使用File.WriteAllBytes()和ReadAllBytes()方法来了解到底如何将Unicode数据写入文件。

动手做！

**1 向文件写一个正常的串，再将其读回。**

使用WriteAllText()方法（前面编写的文本编辑器中也用了这个方法）使第一个按钮将串“Eureka!”写至一个名为“eureka.txt”的文件。然后创建一个新的字节数组eurekaBytes，将文件读入这个数组，再显示所读的全部字节：

```
File.WriteAllText("eureka.txt", "Eureka!");
byte[] eurekaBytes = File.ReadAllBytes("eureka.txt");
foreach (byte b in eurekaBytes)
    Console.Write("{0} ", b);
Console.WriteLine();
```

ReadAllBytes()方法返回一个新字节数组的引用，这个数组中包含从文件读入的所有字节。

你会看到向输出写入了以下字节：69 117 114 101 107 97 33。下面在这一章前面编写的简单文本编辑器（Simple Text Editor）中打开这个文件，会显示“Eureka!”。

**2 让第二个按钮将字节显示为十六进制数。**

并不只有字符映射表能够以十六进制显示数字。几乎所有处理编码数据的程序都会以十六进制显示数据，所以要知道如何进行处理。程序中第二个按钮的事件处理程序代码与第一个按钮的代码基本相同，不过修改Console.Write()那一行代码，替换为以下代码：

```
Console.Write("{0:x2} ", b);
```

这行代码告诉Write()将参数0（输出串后面的第一个参数）显示为一个2字符的十六进制码。所以它会用十六进制写7个字节（45 75 72 65 6b 61 21）而不是十进制数。

十六进制使用数字0到9和字母A到F来表示十六进制的数，所以6B就等于107。

**3 让第3个按钮写希伯来字母。**

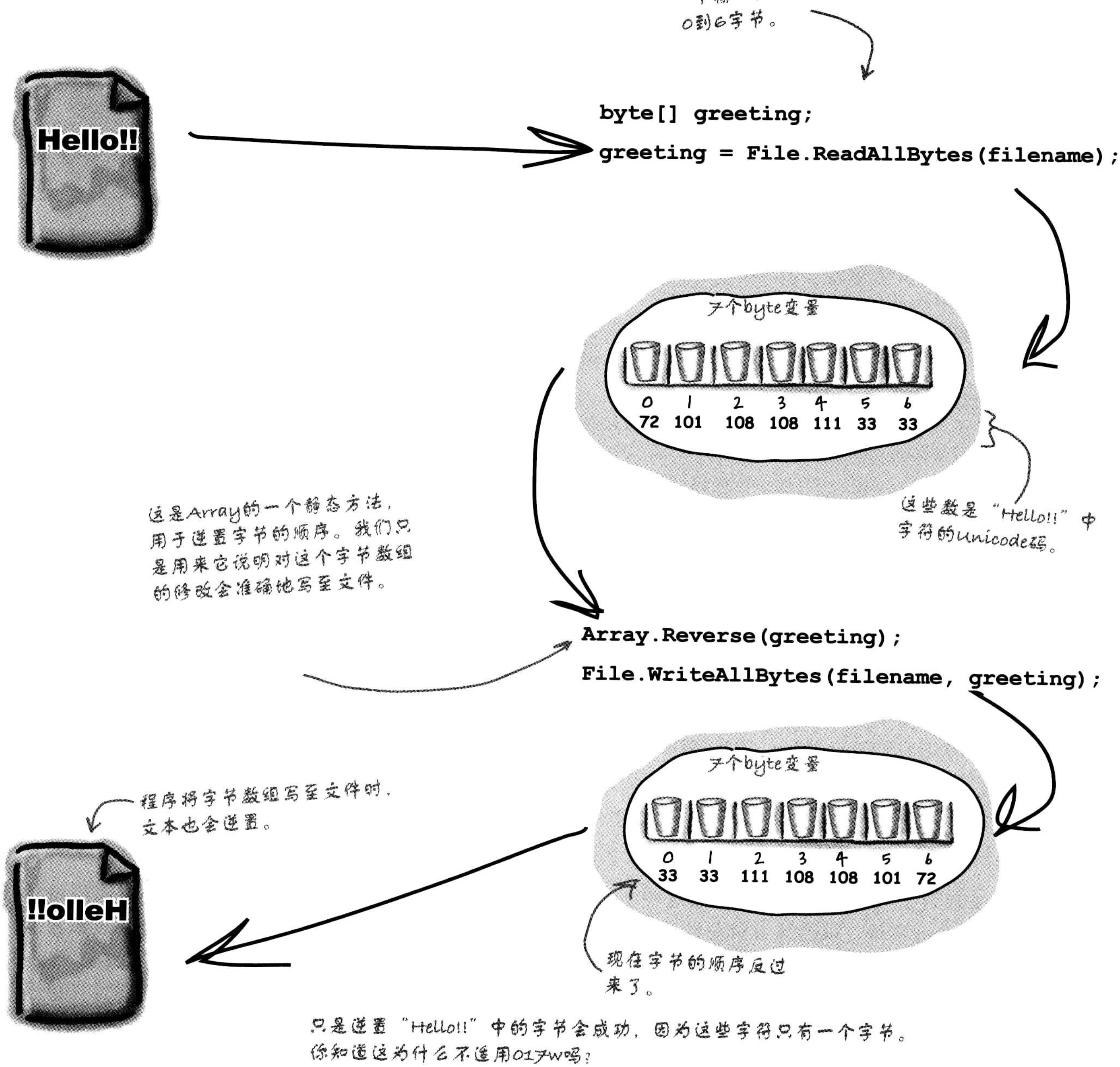
回到字符映射表，双击shin字符（或点击Select按钮）。它会增加到“Characters to copy”框。然后对“Shalom”（希伯来语：喂，你好）中的其余字母做同样的处理（包括Lamed (U+05DC)，Vav (U+05D5)和Final Mem (U+05DD)）。下面为第3个按钮事件处理程序增加代码。看上去它与第2个按钮的代码很相似，不过有一个改变。点击字符映射表中的“Copy”按钮，然后粘贴这些字母替换掉“Eureka!”，并增加Encoding.Unicode参数，代码如下：

```
File.WriteAllText("eureka.txt", "017w", Encoding.Unicode);
```

注意到了吗？IDE是反向粘贴这些字母的！这是因为希伯来语从右向左读，所以只要遇到希伯来Unicode字母，就会从右向左显示。光标放在这些字母中间时，左右箭头按键的作用会反过来！这样键入希伯来文字时就会容易得多。下面运行这个代码，仔细查看输出：ff fe e9 05 dc 05 d5 05 dd 05。前两个字符是“FF FE”，在Unicode中是指这是一个包含2字节字符的串。余下的字节就是希伯来字母，不过它们的顺序颠倒了，所以U+05E9显示为**e9 05**。下面再在Simple Text Editor中打开这个文件，看上去一切正常！

# C#可以使用字节数组移动数据

由于所有数据最后都会编码为字节，所以把文件想成是一个很大的字节数组是有道理的。你已经知道了如何读写字节数组。



# 使用BinaryWriter写二进制数据

StreamWriter还会对数据编码。它专门用于文本和文本编码。

将string、char、int和float写入文件之前都可以先编码为字节数组，但是这相当麻烦。正是因为这个原因，.NET提供了一个很有用的类，名为**BinaryWriter**，它会自动地对数据进行编码，并写至一个文件。你要做的只是创建一个FileStream，把它传入BinaryWriter的构造函数。然后可以调用BinaryWriter的方法来写数据。下面向程序增加另一个按钮，我们将介绍如何使用BinaryWriter。

**动手做！**

1. 先创建一个控制台应用，建立一些要写到文件的数据。

```
int intValue = 48769414;
string stringValue = "Hello!";
byte[] byteArray = { 47, 129, 0, 116 };
float floatValue = 491.695F;
char charValue = 'E';
```

使用File.Create()时，这会创建一个新文件，如果已经有这样一个文件，这个方法会把原文件删除而创建一个全新的文件。另外还有一个File.OpenWrite()方法，它会打开原来的文件，并从头开始覆盖。

2. 要使用BinaryWriter，首先需要用File.Create()打开一个新的流：

```
using (FileStream output = File.Create("binarydata.dat"))
using (BinaryWriter writer = new BinaryWriter(output)) {
```

3. 只需调用它的Write()方法。每次调用这个方法时，都会向文件末尾增加一些新字节，其中包含作为参数传入的数据，而且这些数据已经编码。

```
    writer.Write(intValue);
    writer.Write(stringValue);
    writer.Write(byteArray);
    writer.Write(floatValue);
    writer.Write(charValue);
}
```

每个Write()语句都会把一个值编码为字节，然后将这些字节发送到FileStream对象。可以传入任何值类型，它会自动完成编码。

FileStream将字节写至文件末尾。

**Sharpen your pencil**

4. 下面使用前面的代码读取刚才写的文件。

```
byte[] dataWritten = File.ReadAllBytes("binarydata.dat");
foreach (byte b in dataWritten)
    Console.Write("{0:x2} ", b);
Console.WriteLine(" - {0} bytes", dataWritten.Length);

Console.ReadKey();
```

提示：string可以有不同的长度，所以string最前面必须有一个数告诉.NET这个串有多长。另外，可以使用字符映射表查看串和字符的Unicode值。

把输出写在下面的空上。你能得出对应这5个Write()语句的字节分别是什么吗？用相应的变量名标志每组字节。

__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ - ___ 字节

**Sharpen your pencil Solution**

float和int值写至文件时占4个字节。如果使用long或double，它们会分别占8个字节。

86 29 e8 02 06 48 65 6c 6c 6f 21 2f 81 00 74 f6 d8 f5 43 45 20 bytes

intValue　stringValue　byteArray　floatValue　charValue

string中的第一个字节是6，这是这个串的长度。可以使用字符映射表查找"Hello!"中的各个字符的unicode码，从U+0048开始，最后一个是U+0021。

如果使用Windows计算器将这些字节从十六进制转换为十进制，可以看到它们正是byteArray中的数。

char存放一个unicode字符，'E'只占一个字节，编码为U+0045。

## 使用BinaryReader读回数据

BinaryReader类的工作与BinaryWriter类似。创建一个流，将BinaryReader对象关联到这个流，然后调用它的方法。不过阅读器不知道文件里有哪些数据！而且也没办法知道。float值491.695F编码为d8 f5 43 45。但是这些字节同样可以表示为一个完全合法的int值1 140 185 334。所以需要告诉BinaryReader究竟从文件读什么类型。再向窗体增加一个按钮，让它读取刚才所写的数据。

不要只听我们讲。把读float的那行代码换成一个ReadInt32()调用。（需要把floatRead的类型改为int）。然后可以自己看看从文件读到了什么。

1. 先创建FileStream和BinaryReader对象：

```
using (FileStream input = File.OpenRead("binarydata.dat"))
using (BinaryReader reader = new BinaryReader(input)) {
```

2. 通过调用BinaryReader的不同方法告诉它要读什么类型的数据。

```
int intRead = reader.ReadInt32();
string stringRead = reader.ReadString();
byte[] byteArrayRead = reader.ReadBytes(4);
float floatRead = reader.ReadSingle();
char charRead = reader.ReadChar();
```

对应每个值类型，BinaryReader中都有一个相应的方法返回正确类型的数据。大多数方法都不需要任何参数，不过ReadBytes()需要一个参数告诉BinaryReader要读多少个字节。

3. 通过Console类向控制台输出结果。

```
    Console.Write("int: {0}  string: {1}  bytes: ", intRead, stringRead);
    foreach (byte b in byteArrayRead)
        Console.Write("{0} ", b);
    Console.Write(" float: {0}  char: {1} ", floatRead, charRead);
}
Console.ReadKey();
```

控制台显示的输出如下：

```
int: 48769414  string: Hello!  bytes: 47 129 0 116  float: 491.695  char: E
```

# 还可以手动读写串行化文件

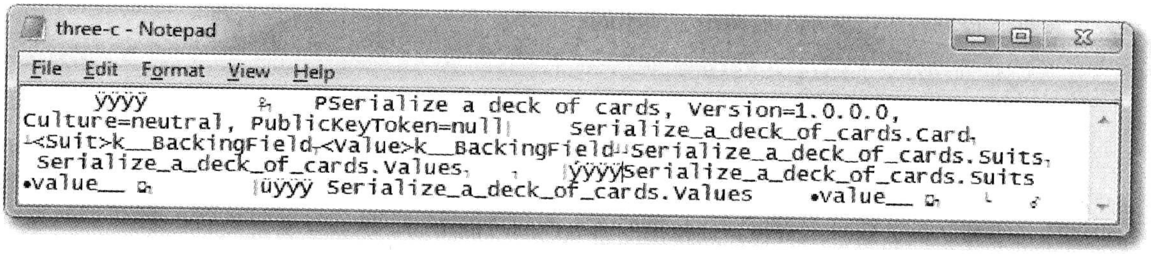
串行化文件在记事本中打开时看上去可不怎么样。你会发现你写的所有文件都在工程的“bin/Debug”文件夹下——下面花点时间进一步了解串行化文件的内部原理。

**动手做！**

**1 把两个Card对象串行化到不同的文件。**

使用前面写的串行化代码将Three of Clubs（梅花3）写到three-c.dat，并把Six of Hearts（红桃6）写到six-h.dat。检查这两个文件确实已经写出，现在在同一个文件夹中，它们的文件大小都相同。然后在记事本中打开其中一个文件。

这个文件里有一些词，但大多不可读。




**2 编写一个循环比较这两个二进制文件。**

这里使用ReadByte()方法从流读下一个字节，它会返回一个int，其中包含这个字节的值。我们还将使用流的Length字段，确保读完整个文件。

```
byte[] firstFile = File.ReadAllBytes("three-c.dat");
byte[] secondFile = File.ReadAllBytes("six-h.dat");
for (int i = 0; i < firstFile.Length; i++)
    if (firstFile[i] != secondFile[i])
        Console.WriteLine("Byte #{0}: {1} versus {2}",
            i, firstFile[i], secondFile[i]);
```

这个循环检查这两个文件的第1个字节，进行比较，然后比较第2个字节、第3个字节等等。发现不同时，向控制台写一行文本。

将这两个文件读到两个不同的字节数组，以便逐字节地比较。由于这是把同一个类串行化到两个不同的文件，所以它们基本上是相同的……不过下面来看究竟有多相似。

**Watch it!**

**写文件时，并不一定总从一张白纸开始！**

如果使用File.OpenWrite()就要当心了。它不会删除文件，只是从开始处覆盖原来的数据。正是因为这个原因，我们一直都使用File.Create()，它会创建一个新文件。

未完，翻到下一页！

# 找出文件差别在哪里，并使用这些信息来修改文件

前面写的循环准确地指出了两个串行化Card文件在哪里存在差异。由于这两个对象只是Suit和Value字段有所不同，所以这应该也是相应文件的唯一差别。因此，如果找到包含花色和牌面大小的字节，就应该能修改这些字节，按我们希望的花色和大小创建一个新牌！

**还可以将对象串行化为XML。翻到附录“其他”中的第9项来了解更多有关内容。**

**3 来看控制台的输出，检查两个文件有什么不同。**

控制台应当显示出有两个字节不同：

```
Byte #322: 1 versus 3
Byte #382: 3 versus 6
```

这很有道理！回过头来看上一章的Suits enum，你会发现Clubs的值正是1，面Hearts的值是3，所以这是第1个差别。再看第2个差别，6和3，这显然就是牌面大小。你看到的字节数可能与这里不同，这并不奇怪：你使用的命名空间可能不同，这会改变文件的长度。

还记得吧，命名空间要作为串行化文件的一部分。如果你的命名空间与此不同，那么字节数自然也不同。

嗯，如果串行化文件中的#322字节表示花色，那我们就能读入这个文件，更改这个字节，再把它写回文件，这样就能改变牌的花色了（要记住，你自己的串行化文件中花色可能存储在不同的位置上）。

**4 编写代码，手动创建一个包含King of Spades（黑桃K）的新文件**

取读入的一个数组，修改这个数组，让它包含一张新牌，再把它写回文件。

```
firstFile[322] = (byte)Suits.Spades;
firstFile[382] = (byte)Values.King;
File.Delete("king-s.dat");
File.WriteAllBytes("king-s.dat", firstFile);
```

如果你发现第3步中有不同的字节数，这里要替换为你的字节数。

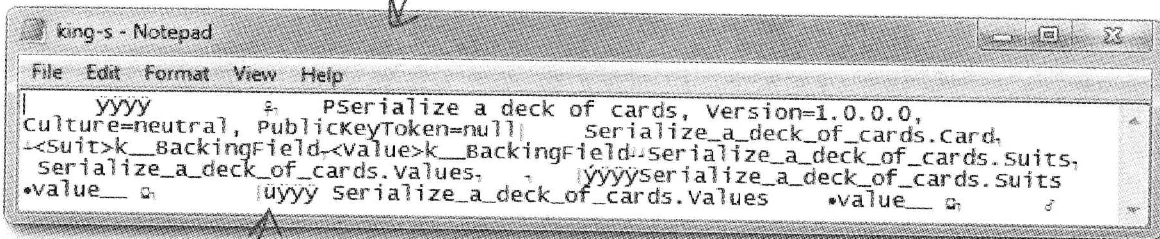
现在从**king-s.dat**逆串行化扑克牌，看看它是不是黑桃K！

既然你知道哪些字节包含花色和牌面大小，将数组写到king-s.dat之前可以只更改数组中的这些字节。

# 处理二进制文件可能很麻烦

如果你有一个文件但不能确定里面有什么，该怎么做呢？你不知道是哪个应用创建了这个文件，你需要对它有所了解，但是在记事本中打开这个文件时，看起来只像一堆垃圾。如果没有其他办法，而你确实想知道里面是什么，你会怎么做？请看下面的图，很显然，记事本不算一个合适的工具。

这是串行化的牌，在记事本中打开后，可以看到没有任何意义。

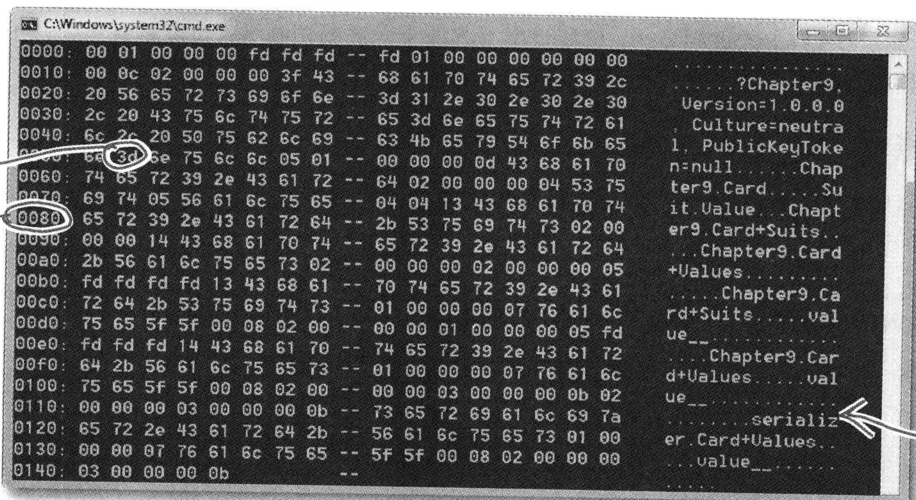



可以发现一些东西——比如枚举名（"Suit"和"Value"），还有我们使用的命名空间名（"Serialize_a_deck_of_cards"）。但是这些并没有太大帮助。

还有一个办法，利用一种称为"十六进制转储（hex dump）"的格式，这是一种查看二进制数据的标准方法。与在记事本中查看文件相比，这种方法能提供更多信息。十六进制是显示文件中字节的一种便利方法。每个字节用十六进制显示占两个字符，所以可以在很小的空间里查看很多数据，而且利用这种格式能很容易地发现规律。另外，如果按每行8，16或32字节来显示二进制数据会很有好处，因为大多数二进制数据都划分为4、8、16或32字节的块……C#中的所有类型就是如此。例如，一个int占4个字节，串行化到磁盘时就是4字节长。以下还是原来的文件，这里使用了Windows提供的一个免费十六进制转储程序，以十六进制转储格式来显示：

可以立即看到文件中各字节的数字值。

每一行开头的数是这一行第一个字节的偏移量（或距文件起始位置的距离）。




可以看到原来的文本，不过extended字符已经替换为点号（.）。

# 使用文件流构建一个十六进制转储工具

十六进制转储格式是文件内容的一个十六进制视图，这是程序员仔细查看文件内部结构的一种很常用的方法。大多数操作系统都提供了内置的十六进制转储工具。遗憾的是，Windows没有这样的内置工具。所以我们来建一个！

## 如何建立十六进制转储格式

先从我们熟悉的一些文本开始：

```
We the People of the United States, in Order to form a more perfect Union...
```

以下是这个文本的十六进制转储格式：

再次指出，可以立即看到文件中各个字节的数字值。

```
0000: 57 65 20 74 68 65 20 50 -- 65 6f 70 6c 65 20 6f 66   We the People of
0010: 20 74 68 65 20 55 6e 69 -- 74 65 64 20 53 74 61 74    the United Stat
0020: 65 73 2c 20 69 6e 20 4f -- 72 64 65 72 20 74 6f 20   es, in Order to
0030: 66 6f 72 6d 20 61 20 6d -- 6f 72 65 20 70 65 72 66   form a more perf
0040: 65 63 74 20 55 6e 69 6f -- 6e 2e 2e 2e               ect Union...
```

增加各行最前面的数字，这里使用一行中第一个字节的偏移量。

另外，需要把垃圾字符替换成点号。

所有这些数字57，65，6F，都是文件中一个字节的值。之所以某些“数字”的值是字母，原因在于它们是十六进制数。这是写数的另一种方法。采用十六进制时，不是使用从0到9的10个数，而使用从0到9再加上字母A到F的16个十六进制数。

十六进制转储格式中的每一行表示输入文件中的16个字符（我们要使用这个输入文件生成这个转储数据）。在我们的转储格式中，前4个字符是文件中的偏移量，第一行从0开始，下一行从16(或十六进制数10)开始，然后是32 (十六进制为20), 依此类推（其他十六进制转储格式可能稍有不同，不过我们使用的是上述格式）。

## 处理十六进制

可以把十六进制数直接放在你的程序中，只是要在数字前加上字符0x（0后面加一个x）：

```
int j = 0x20;
MessageBox.Show("The value is " + j);
```

使用+操作符将一个数与一个串连接时，它会转换为十进制数。可以使用静态的`String.Format()`方法把数字转换为一个十六进制格式的串：

```
string h = String.Format("{0:x2}", j);
```

String.Format()使用的参数与Console.WriteLine()所用参数类似，所以不需要学习新内容就可以使用这个方法。

# StreamReader和StreamWriter可以胜任（对目前而言）

我们的十六进制转储工具可以把转储数据写到一个文件，另外，由于它只是写文本，所以StreamWriter就能胜任。不过我们还可以利用StreamReader中的**ReadBlock()**方法。它能把一个字符块读入char数组（需要指定想读取的字符个数），这个方法会读入指定数目的字符，如果文件中剩下的字符不足（少于指定的字符个数），它会读入文件中余下的所有字符。由于我们每行显示16个字符，所以这里将读取16个字符的字符块。

这个方法之所以名为"ReadBlock()"，这是因为调用这个方法时，它会"阻塞"（这说明它会一直执行，不会返回到你的程序），直到读完你要求的所有字符，或者读完剩余的所有数据。

下面再为你的程序增加一个按钮，加入这个十六进制转储工具。修改前两行，让它们指向你自己的硬盘上的文件。先尝试查看一个串行化Card文件。然后看能不能修改这个程序来使用Open和Save As对话框（而不是直接指定文件名）。

```
using (StreamReader reader = new StreamReader(@"c:\files\inputFile.txt"))
using (StreamWriter writer = new StreamWriter(@"c:\files\outputFile.txt", false))
{
  int position = 0;

  while (!reader.EndOfStream) {
     char[] buffer = new char[16];
     int charactersRead = reader.ReadBlock(buffer, 0, 16);
     writer.Write("{0}: ", String.Format("{0:x4}", position));
     position += charactersRead;

     for (int i = 0; i < 16; i++) {
        if (i < charactersRead) {
           string hex = String.Format("{0:x2}", (byte)buffer[i]);
           writer.Write(hex + " ");
        }
        else
           writer.Write(" ");

       if (i == 7) { writer.Write("-- "); }
       if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = '.'; }
     }
     string bufferContents = new string(buffer);
     writer.WriteLine(" " + bufferContents.Substring(0, charactersRead));
  }
}
```

如果文件中还有未读的字符，StreamReader的EndofStream属性会返回false。

这个ReadBlock()调用会向一个char数组最多读入16个字符。

静态String.Format方法将数字转换为串。"{0:x4}"告诉Format()要把第2个参数（在这里就是position）输出为一个4字符的十六进制数。

这个循环处理各个字符，并把它们逐个打印到输出中的一行上。

有些字符的值小于32，这些字符不能打印，所以它们都用一个点号代替。

把一个char[]数组传入string的重载构造函数，可以把它转换为一个string。

每个串都有一个Substring方法，可以返回这个串的一部分。在这里，它会返回从起始位置（位置0）开始的前charactersRead个字符（请注意这个循环最前面如何设置charactersRead ReadBlock()方法会返回读入数组的字符个数）。

# 建立更好的十六进制转储工具
# 使用Stream.Read()从流读取字节

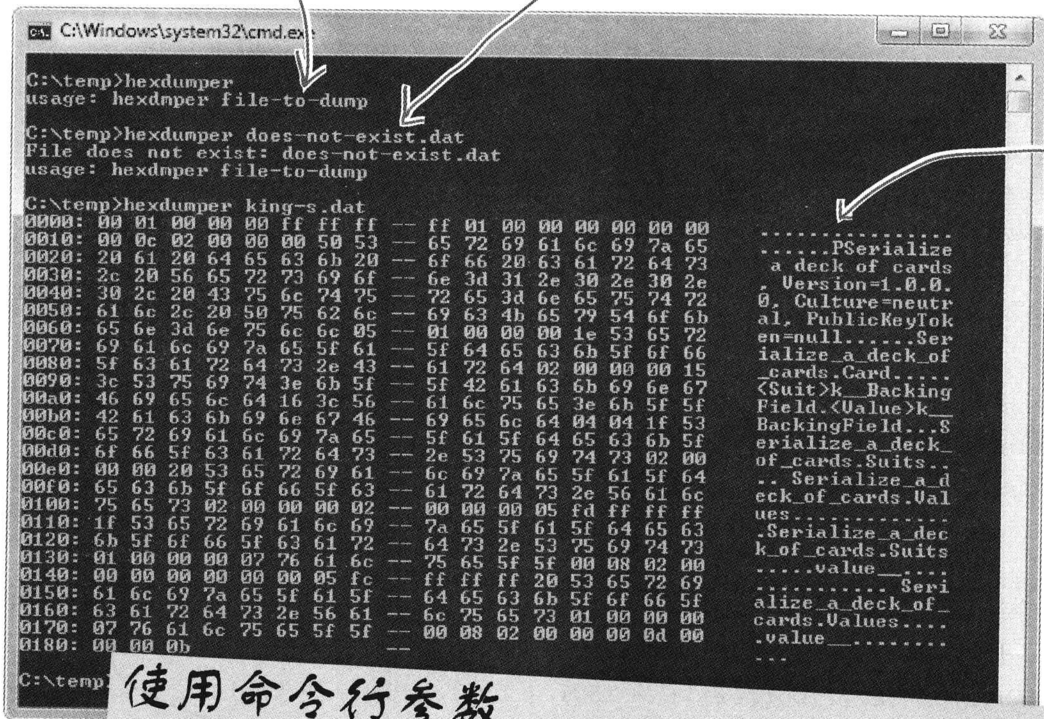
动手做

这个十六进制转储工具只适用于文本文件。不过这里存在一个问题。可以试着使用`File.WriteAllBytes()`将一个字节数组（包含大于127的值）写至一个文件，然后在你的转储工具中运行。唉呀，它们统统都会读入为“fd”!这是因为StreamReader只是用来读取文本文件，其中只能包含值小于128的字节。所以下面来修改这个程序，使用**`Stream.Read()`**方法从流直接读取字节。作为额外奖励，我们要把它构建得像一个真正的十六进制转储工具：要取一个文件名作为命令行参数。

创建一个新的控制台应用，命名为**`hexdumper`**。程序的代码见下一页。运行这个程序会看到以下控制台窗口：

如果运行hexdumper而没有提供任何参数，它会返回一个错误消息，并以一个错误码退出。

如果传入一个文件名，但指定文件不存在，也会以一个错误退出。




如果传入一个合法的文件名，它会把文件内容的十六进制转储写到控制台。

正常情况下，我们使用Console.WriteLine()写至控制台。但是我们会使用Console.Error.WriteLine()写错误消息，这样一来，如果我们使用了>或>>重定向输出，这些错误消息不会重定向。

## 使用命令行参数

每次创建一个新的控制台应用工程时，Visual Studio会创建一个Program类，它的入口点方法声明为：static void Main(string[] args)。如果运行程序时提供了命令行参数，args参数会包含这些命令行参数。并不只是控制台应用如此：打开任何Windows Forms应用工程的Program.cs文件，也会看到同样的声明。

调试程序时可能希望传入命令行参数。要在IDE的调试工具中运行程序时传入参数，可以从“工程”菜单选择“Properties…”，在Debug标签页中输入这些参数。

使用args参数传递命令行参数。

如果args.Length不等于1，则可能为0，也可能在命令行传入了多个参数。

```
static void Main(string[] args)
{
    if (args.Length != 1)
    {
        Console.Error.WriteLine("usage: hexdmper file-to-dump");
        System.Environment.Exit(1);
    }
    if (!File.Exists(args[0]))
    {
        Console.Error.WriteLine("File does not exist: {0}", args[0]);
        System.Environment.Exit(2);
    }
    using (Stream input = File.OpenRead(args[0]))
    {
        int position = 0;
        byte[] buffer = new byte[16];
        while (position < input.Length)
        {
            int charactersRead = input.Read(buffer, 0, buffer.Length);
            if (charactersRead > 0)
            {
                Console.Write("{0}: ", String.Format("{0:x4}", position));
                position += charactersRead;

                for (int i = 0; i < 16; i++)
                {
                    if (i < charactersRead)
                    {
                        string hex = String.Format("{0:x2}", (byte)buffer[i]);
                        Console.Write(hex + " ");
                    }
                    else
                        Console.Write("   ");

                    if (i == 7)
                        Console.Write("-- ");

                    if (buffer[i] < 32 || buffer[i] > 250) { buffer[i] = (byte)'.'; }
                }
                string bufferContents = Encoding.UTF8.GetString(buffer);
                Console.WriteLine("   " + bufferContents.Substring(0, charactersRead));
            }
        }
    }
}
```

注意这里如何使用Console.Error.WriteLine()。

这个Exit()方法会让程序退出。如果把它传至一个int，将返回这个错误码（编写命令脚本和批文件时这很有用）。

下面确保传入了一个合法的文件。如果文件不存在，打印一个不同的错误消息，并返回一个不同的错误码。

我们不需要StreamReader，因为这里直接从流读取字节。

使用Stream.Read()方法将字节直接读入一个缓冲区buffer。注意这一次buffer是一个字节数组。这是有道理的，我们在读取字节，而不是从文本文件读取字符。

程序的这一部分与前面完全相同，只不过buffer包含的是字节而不是字符（不过不论哪种情况String.Format()都能正常工作）。

这是一种将字节数组转换为串的简便方法。这是Encoding.UTF8的一个方法（另一个Unicode编码，或ASCII编码），因为不同的编码可能把相同的字节数组映射到不同的串。

# there are no Dumb Questions

**问：为什么使用了File.ReadAllText()和File.WriteAllText()之后不需要使用Close()方法来关闭文件？**

**答：** File类有许多非常有用的静态方法，它们会自动打开一个文件、读写数据，然后自动将文件关闭。除了ReadAllText() 和WriteAllText()方法，还有处理字节数组的ReadAllBytes()和WriteAllBytes()方法，读写串数组的ReadAllLines()和WriteAllLines()方法（数组中的各个串是文件中的各行），它们也有这个特点。所以这些方法都会自动打开和关闭流，所以只需一条语句就能完成全部文件操作。

**问：既然FileStream有完成读写的方法，为什么还需要使用StreamReader和StreamWriter呢？**

**答：** FileStream类对于向二进制文件读写字节确实非常有用。它的读写方法处理的是字节和字节数组。但是很多程序都会大量使用文本文件，如第一个版本的借口生成系统，它只是将串写到文件中。对于这些情况，StreamReader和StreamWriter就派上用场了。它们的方法专门为读写文本行而设计。如果没有这些类，倘若你想从一个文件读入文本行，就必须首先读一个字节数组，然后编写一个循环在这个数组中搜索换行符，显而易见，有了StreamReader和StreamWriter这两个类，你的日子会好过得多。

**问：什么情况下使用File，而什么情况下要使用FileInfo？**

**答：** File和FileInfo类的主要区别在于，File中的方法是静态的，所以不必创建File的实例就能调用这些方法。另一方面，FileInfo要求必须用一个文件名来实例化对象。某些情况下，这可能要麻烦一些，比如你只需要完成一个文件操作（如只是删除或移动一个文件）。但另一方面，如果需要对同一个文件完成多个文件操作，那么FileInfo会更方便，因为只需向它传递一次文件名。你要根据遇到的具体情况来决定使用哪一个类。换句话说，如果你要完成一个文件操作，可以使用File。如果要接连完成很多文件操作，那么建议使用FileInfo。

**问：请等等。为什么写“Eureka!”时每个字符占一个字节，而我写希伯来文字时它们却占2个字节？另外这些字节最前面的“FF FE”是什么意思？**

**答：** 你看到的正是两种紧密相关的Unicode编码之间的差别。无格式的英语字母、数字、普通的标点符号，以及一些标准字符（如大括号、&和你在键盘上看到的其他字符）的Unicode码都很小（0～127之间）。如果你之前用过ASCII码，应该知道这与ASCII字符相同。如果一个文件只包含这些Unicode值很小的字符，就会直接输出其字节（只占一个字节）。但是如果增加了Unicode值更大的字符，问题就要复杂一些了。一个字节只能存放0～255之间的数。但是连续两个字节可以存储0～65536（十六进制表示为FFFF）之间的数。程序打开一个文件时，文件应该能够告诉程序它包含这些Unicode码较大的字符。所以它会在文件最前面放置一个特殊的保留字节：“FF FE”。这称为“字节顺序标志”。一旦程序看到这个标志，就能知道所有字符编码分别都占两个字节（所以E会编码为00　45，加了前导0）。

**问：为什么称它是字节顺序标志？**

**答：** 还记得字节是怎么逆置的吗？Shin的Unicode值U+05E9写到文件中时变成了E9　05。这称为“little endian”顺序。再来看写这些字节的代码，向WriteAllText()再增加第3个参数：Encoding.BigEndianUnicode。这会告诉它以“big endian”顺序写数据，这样就不会颠倒字节的顺序了。这一次你会看到字节写为“05 E9”，另外还会看到字节顺序标志也有了变化，变成了“FE FF”。你的Simple Text Editor很聪明，这两种顺序都能读！

**如果你在写一个串，其中只包含值很小的Unicode字符，写出时每个字符占一个字节。但是如果其中有值很大的Unicode字符，写出时每个字符会占两个字节。**

这种编码称为UTF-8，这也是.NET默认使用的编码。通过传入一个不同的Encoding值，可以要求File.WriteAllText()使用一种不同的编码。有关Unicode编码的更多内容请见http://unicode.org。




修改Brian的借口生成系统，使用包含串行化Excuse对象的二进制文件，而不是文本文件。

1. **使Excuse类可串行化。**
用`[Serializable]`属性标志`Excuse`类，使它可以串行化。另外，需要增加一行`using`代码：

```
using System.Runtime.Serialization.Formatters.Binary;
```

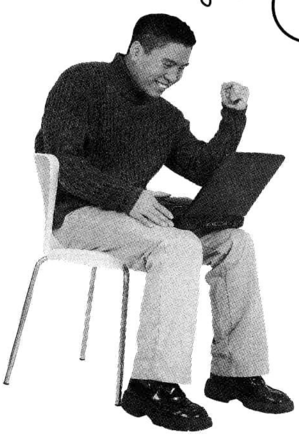
2. **修改Excuse.Save()方法，串行化借口对象。**
`Save()`方法向文件夹写一个文件时，不再使用`StreamWriter`写文件，而是打开一个文件自行串行化。你要知道当前类如何自行逆串行化。

提示：在类中使用哪个关键字返回它自己的一个引用？

3. **修改Excuse.OpenFile()方法逆串行化一个借口。**
需要创建一个临时`Excuse`对象从文件逆串行化，然后将临时对象的字段复制到当前类对象。

4. **接下来只需修改窗体，使用一个新的文件扩展名。**
只需对窗体做一个很小的修改。由于我们不再处理文本文件，所以不应当使用`.txt`扩展名。修改对话框默认文件名和目录搜索代码来处理`*.excuse`文件。

太棒了，真是太简单！保存和打开借口的所有代码都在Excuse类里，我只需要修改这个类，几乎不用去碰窗体。看起来，窗体甚至不关心类是如何存储数据的。它只是传入文件名，就能相信一切都会妥当地保存。



**非常正确！你的代码很容易修改，因为类得到了很好的封装。**

如果一个类隐藏了它的内部操作，不允许程序的其他部分访问，而只是公布确实需要公布的行为，这就是一个封装得很好的类。在借口管理系统程序中，窗体对于借口如何保存为文件并不了解，它不知道任何有关信息。窗体只是向借口类传入一个文件名，借口类自会负责完成其余的工作。这样一来，如果类处理文件的方式要做很大的改动也很容易。类封装得越好，以后修改时就越容易。

还记得4大OOP原则中的封装原则吗？使用这些原则可以让程序更出色，这就是一个很好的例子。

**Exercise Solution**

修改Brian的借口生成系统，使用包含串行化Excuse对象的二进制文件，而不是文本文件。

只需修改窗体中的这3个语句：save按钮Click事件处理程序中的两个语句，以及open按钮Click事件处理程序中的一个语句，这里只是修改对话框来使用.excuse扩展名。

```
private void save_Click(object sender, EventArgs e) {
  // existing code
  saveFileDialog1.Filter = "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*" ;
  saveFileDialog1.FileName = description.Text + ".excuse" ;
  // existing code
}
private void open_Click(object sender, EventArgs e) {
  // existing code
  openFileDialog1.Filter =
     "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*" ;
  // existing code
}
```

这里使用标准的保存和打开对话框。

对窗体只做了一点修改：改变了传入Excuse类的文件扩展名。

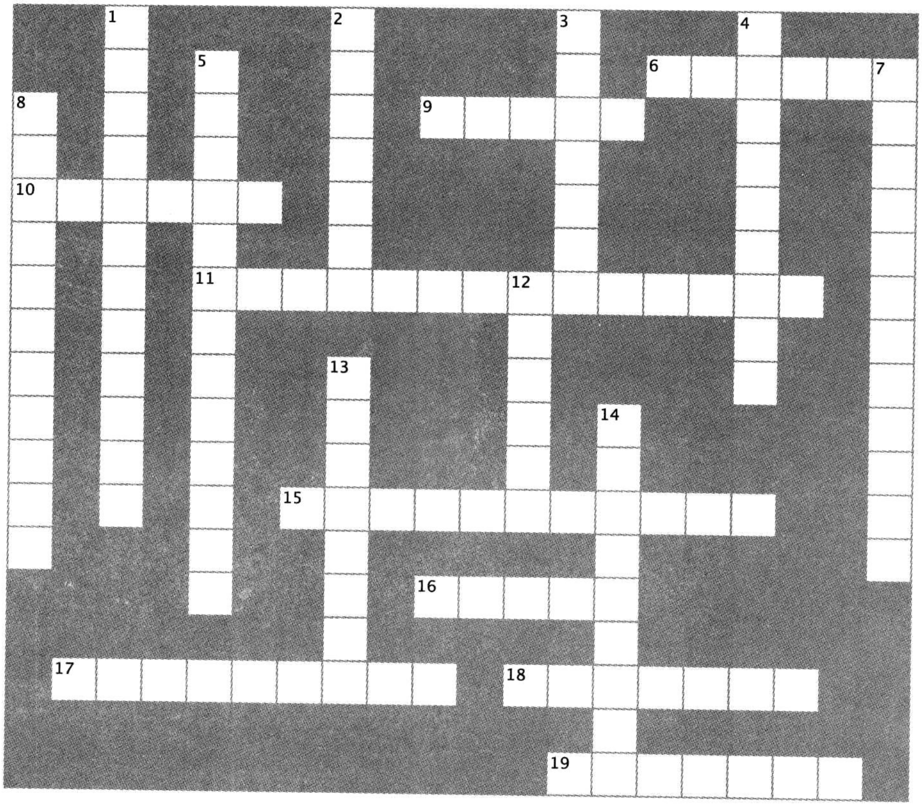
这是完整的Excuse类。

```
[Serializable]
class Excuse {
    public string Description { get; set; }
    public string Results { get; set; }
    public DateTime LastUsed { get; set; }
    public string ExcusePath { get; set; }
    public Excuse() {
        ExcusePath = "";
    }
    public Excuse(string excusePath) {
        OpenFile(excusePath);
    }
    public Excuse(Random random, string folder) {
        string[] fileNames = Directory.GetFiles(folder, "*.excuse");
        OpenFile(fileNames[random.Next(fileNames.Length)]);
    }
    private void OpenFile(string excusePath) {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath)) {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    public void Save(string fileName) {
        BinaryFormatter formatter = new BinaryFormatter();
        using (Stream output = File.OpenWrite(fileName)) {
            formatter.Serialize(output, this);
        }
    }
}
```

加载随机借口的构造函数需要寻找".excuse"而不是"*.txt"文件。

传入"this"，因为希望这个类串行化。

# 文件填字游戏




## 横向

6. File类中的一个方法，用来检查一个特定文件是否在硬盘上。

9. 这个语句指示switch语句中一个case的结束。

10. FileStream继承的抽象类。

11. 这是一个非可视化控件，能弹出标准Windows“另存为”对话框。

15. 写十六进制数的方式。

16. 如果没有调用这个方法，流就会一直锁定为打开状态，以至于其他方法或程序无法再打开这个流。

17. 这是StreamReader的一个方法，用于将数据读入一个char[]数组。

18. 这种编码系统为每个字符指定一个唯一的数。

19. 当switch中所测试的值与所有case都不匹配时，使用这个语句来指示需要执行哪些语句。

## 纵向

1. 这个类的一个方法可以将一个类型写入文件。

2. 这是Array类中的一个静态方法，可以将一个数组逆置。

3. 用户在输入控件中修改数据时运行的事件处理程序。

4. 这个类有许多静态方法用来处理文件夹。

5. 使用这个OOP原则可以更容易地维护你的代码。

7. 如果不使用这个属性来指示一个类可以写至一个流，BinaryFormatter将生成一个错误。

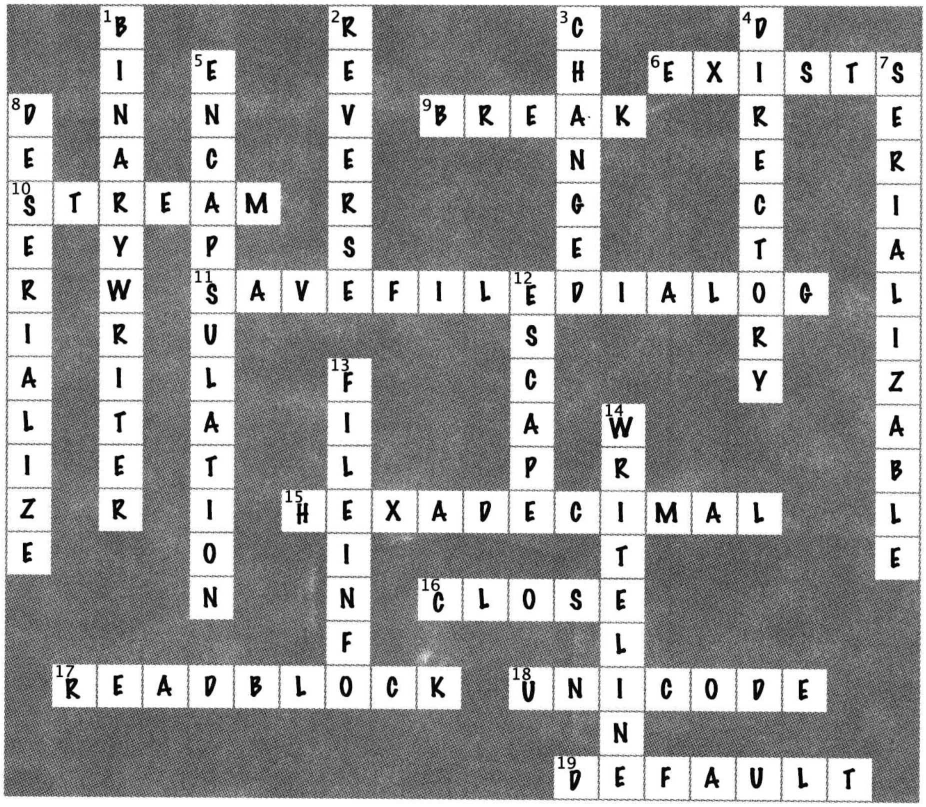
8. 这个BinaryFormatter方法从流读入一个对象。

12. \n和\r 是这种序列的例子。

13. 利用这个类可以对一个特定文件完成File类的所有操作。

14. 这个方法向一个流发送文本，后面加一个换行符。

# 文件填字游戏答案



1 BINARYWRITER
2 REVERSE
3 CHANGE
4 DIRECTORY
5 ENCAPSULATION
6 EXISTS
7 SERIALIZABLE
8 DESERIALIZE
9 BREAK
10 STREAM
11 SAVEFILEDIALOG
12 ESCAPE
13 FILEINF
14 WRITELINE
15 HEXADECIMAL
16 CLOSE
17 READBLOCK
18 UNICODE
19 DEFAULT

*10 异常处理*

# 救火太老套




**程序员可不是消防员。**

你夜以继日地努力工作，遍查各种技术手册，并通读几本绝妙的Head First书，终于攀上了你职业的高峰，成为一名首席程序员，不过还是经常因为你的程序崩溃或者不像预期得那么好而被频频叫回。如果希望编程不那么单调乏味，最好的做法就是去修正一个奇怪的bug……不过，利用异常处理，可以编写代码来处理可能出现的问题。更妙的是，甚至可以一方面处理这些问题，另一方面让程序继续运行。

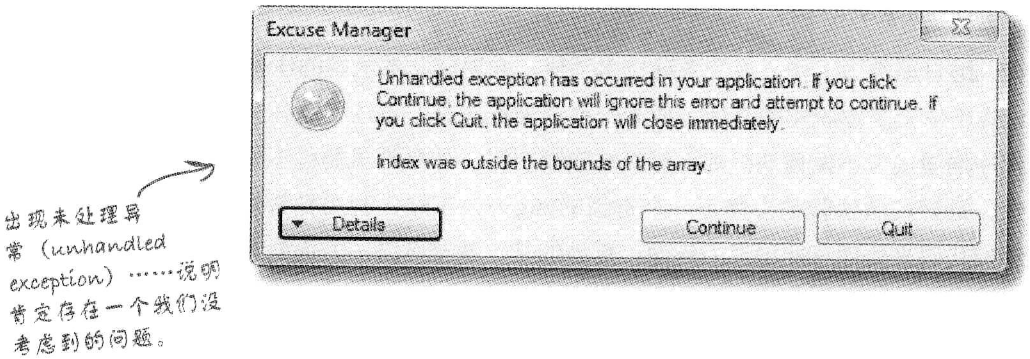
# Brian希望他的借口能移动

Brian最近调职到国际部。现在他得全世界飞来飞去。不过他仍然需要清楚自己用过哪些借口，所以在笔记本电脑里安装了这个程序，以便带着它到任何地方去。




## 但程序无法运行!

Brian点击“Random Excuse”按钮后，却得到一个看上去很麻烦的错误。好像是说无法找到他的借口，这到底是什么意思？



## Sharpen your pencil

这是另外一个例子，这里也是一段无法运行的代码。这个代码抛出了5个不同的异常，错误消息分别显示在右边。你的任务是找出与异常对应的问题代码。请仔细读异常消息，其中给出了很好的提示。

```
public static void BeeProcessor() {
    object myBee = new HoneyBee(36.5, "Zippo");
    float howMuchHoney = (float)myBee;

    HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
    double beeName = double.Parse(anotherBee.MyName);

    double totalHoney = 36.5 + 12.5;
    string beesWeCanFeed = "";
    for (int i = 1; i < (int) totalHoney; i++) {
        beesWeCanFeed += i.ToString();
    }
    float f =
      float.Parse(beesWeCanFeed);

    int drones = 4;
    int queens = 0;
    int dronesPerQueen = drones / queens;

    anotherBee = null;
    if (dronesPerQueen < 10) {
        anotherBee.DoMyJob();
    }
}
```

调用double.Parse("32")会解析一个串("32")，并返回一个double值，如32。

如果一个引用没有指向任何对象，它会得到一个特殊的值null。设置一个引用为null，这就是告诉C#这个引用并未指向任何对象。

**① OverflowException was unhandled**
Value was either too large or too small for a Single.

**② NullReferenceException was unhandled**
Object reference not set to an instance of an object.

**③ InvalidCastException was unhandled**
Specified cast is not valid.

**④ DivideByZeroException was unhandled**
Attempted to divide by zero.
**Troubleshooting tips:**
Make sure the value of the denominator is not zero before performing a division operation.
Get general help for this exception.
Search for more Help Online...

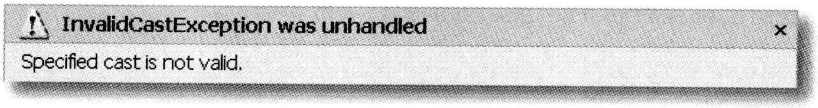
**⑤ FormatException was unhandled**
Input string was not in a correct format.
**Troubleshooting tips:**
Make sure your method arguments are in the right format.
When converting a string to DateTime, parse the string to take the date before putting each variable into the DateTime object.
Get general help for this exception.
Search for more Help Online...

## Sharpen your pencil Solution

你的任务是找出与异常对应的问题代码。

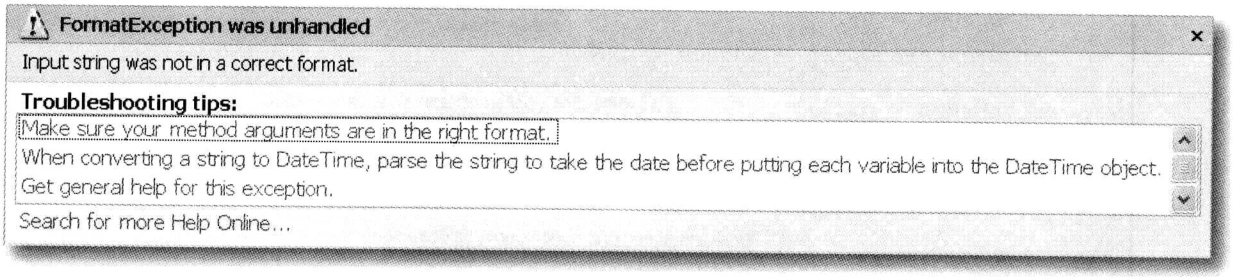
```
object myBee = new HoneyBee(36.5, "Zippo");
float howMuchHoney = (float)myBee;
```

C#允许将myBee强制转换为float，但是没有办法把一个HoneyBee对象转换为float值。代码运行时，CLR完全不知道究竟如何转换，所以会抛出一个InvalidCastException异常。



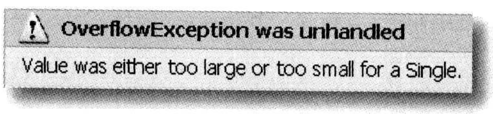

```
HoneyBee anotherBee = new HoneyBee(12.5, "Buzzy");
double beeName = double.Parse(anotherBee.MyName);
```

Parse()方法希望你提供一个特定格式的串，它不知道如何把"Buzzy"串转换为一个数。因此抛出了一个FormatException。




```
double totalHoney = 36.5 + 12.5;
string beesWeCanFeed = "";
for (int i = 1; i < (int) totalHoney; i++) {
    beesWeCanFeed += i.ToString();
}
float f = float.Parse(beesWeCanFeed);
```

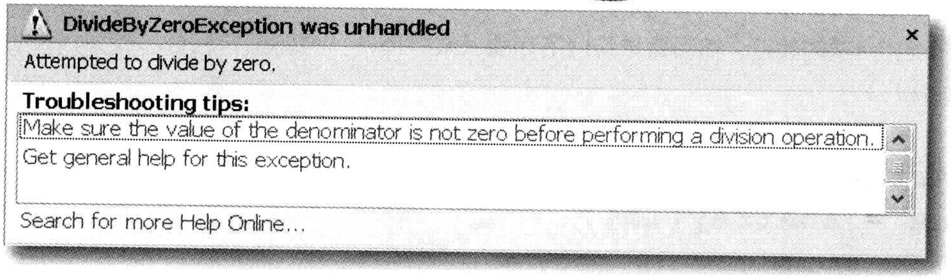
for循环会创建一个名为beesWeCanFeed的串，其中包含的数至少有60位。一个float变量不可能保存这么大的数，把它强制转换为一个float时就会抛出一个OverflowException。




你不会连续地得到所有这些异常。程序抛出第一个异常后就会中止。只有修正了第一个问题后才会得到第二个异常。

```
int drones = 4;
int queens = 0;
int dronesPerQueen = drones / queens;
```

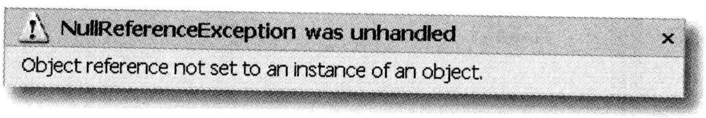
很容易抛出DivideByZeroException。任何数除以0就会得到这个异常。




任何数除以0总会抛出这种异常。即使你不知道queens的值，只要在drones除以queens之前检查queens值并确保它不为0就可以避免这个异常。

```
anotherBee = null;
if (dronesPerQueen < 10) {
    anotherBee.DoMyJob();
}
```

设置anotherBee引用变量等于null，就是在告诉C#，这个引用没有指向任何东西。所以它不指向任何对象，而是什么也没有。C#通过抛出NullReferenceException来告诉你：没有对象可以调用DoMyJob()方法。




**DivideByZero错误本不应该发生。只需查看代码就能看出这里有问题。其他异常也是一样。这些问题都是可以避免的，你对异常了解得越多，就越能保证代码不会崩溃。**

# 程序抛出一个异常时，.NET会生成一个Exception对象

你已经看到了.NET如何告诉你程序中出了问题：它会抛出一个异常（exception）。在.NET中，出现一个异常时，就会创建一个对象来表示出现的问题。毫无疑问，这个对象就叫作`Exception`。

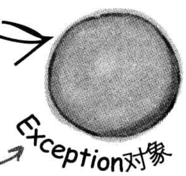
例如，假设有一个包含4项的数组。然后尝试访问第16项（索引15，因为索引从0开始计）：

**ex-cep-tion，名词。**
一个人或事物不符合一般规律或者不遵循某个规则。Jim不喜欢花生酱，不过Ken的花生酱软糖是个例外。

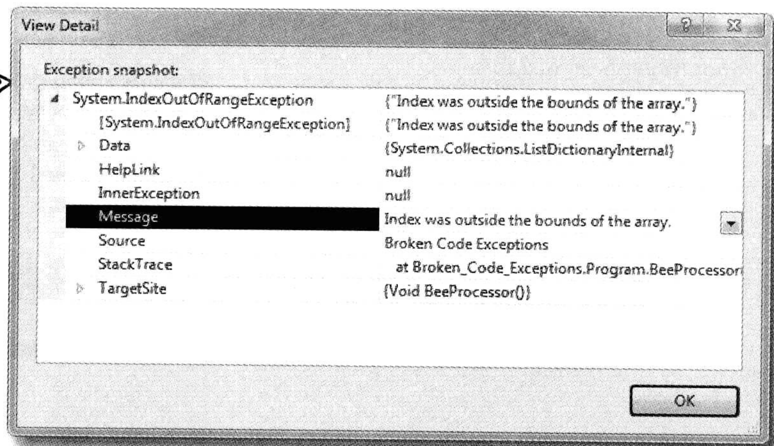
```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```

这个代码显然会带来问题。

一旦程序遇到一个未处理异常，就会生成一个对象，其中包含有关的所有数据。




在未处理异常窗口点击View Detail链接可以看到它的详细信息。




异常对象包含一个消息，指出哪里出了问题，还包含一个系统内存调用列表，可以一直追溯到导致异常的事件。

.NET会负责创建一个对象，因为它希望为你提供导致这个异常的所有相关信息。你可能需要修正一些代码，或者可能只需要对程序中如何处理某个特定的情况做一些修改。

在这里，**`IndexOutOfRangeException`**指示存在一个bug：你想使用一个越界索引访问数组。你还得到了出现问题那行代码的准确位置，这样就能很容易地追踪到问题（即使有数千行代码）。

## there are no Dumb Questions

**问：为什么有这么多种异常？**

**答：** 你可能会写出C#不知道如何处理的代码，而写出这种代码的方式多种多样。如果程序只是提供一个一般性的错误消息（“第37行出现一个问题”），要想排除这个错误就会很困难。如果能针对性地知道出现了哪些类型的错误，跟踪和修正代码中的这些问题就能容易得多。

**问：那么，到底什么是异常呢？**

**答：** 异常就是出现问题时.NET创建的一个对象。你也可以在代码中生成异常（稍后还会更详细地说明）。

**问：等一下，你说什么？异常是一个对象？**

**答：** 没错，异常就是一个对象。这个对象的属性可以告诉你异常的有关信息。例如，它有一个Message属性，其中包含一个很有用的串，如“Specified cast was invalid”（指定强制类型转换不合法）和“Value was either too large or too small for a Single”（值过大或过小），异常窗口就是利用这个串生成的。.NET之所以要生成异常，是为了在执行抛出异常的语句时能尽可能多地提供这个异常的有关信息。

**问：可是，我还是不太清楚。对不起，能不能再解释一下为什么有那么多种异常？**

**答：** 因为你的代码可能会以很多出乎意料的方式执行。所以可能会有很多情况导致你的代码崩溃。如果不知道代码为什么崩溃，要想排除问题就相当困难。通过在不同情况下抛出不同类型的异常，.NET就能为你提供很多有意义的信息，帮助你跟踪并修正问题。

**问：这么说异常是来帮助我的，而不是让我头疼？**

**答：** 确实如此！异常本来是为了帮助你考虑原来没有预料的情况。很多人看到代码抛出异常时都会头疼。不过，如果你把异常想成是.NET帮助你跟踪和调试程序的一种方法，会发现它对于跟踪导致代码出问题的原因确实很有帮助。

**问：那么代码抛出一个异常时，并不一定是因为我做错了，是这样吗？**

**答：** 完全正确。有时数据可能不是你预期的数据，比如你的一个方法要处理一个数组，而这个数组比你原先写方法时所期望的数组长得多或短得多，就会出现问题。另外，不要忘记使用程序的是人，人总会犯错，他们总会以意想不到的方式使用你的程序。而异常正是.NET对此提供的一种方法，可以帮助你处理这些不可预见的情况，使代码仍能平稳运行，而不是简单地崩溃或者给出一个难懂的、毫无意义的错误消息。

**问：一旦知道找什么，就能清楚地看出前一页上的代码会崩溃。是不是所有异常都这么容易发现？**

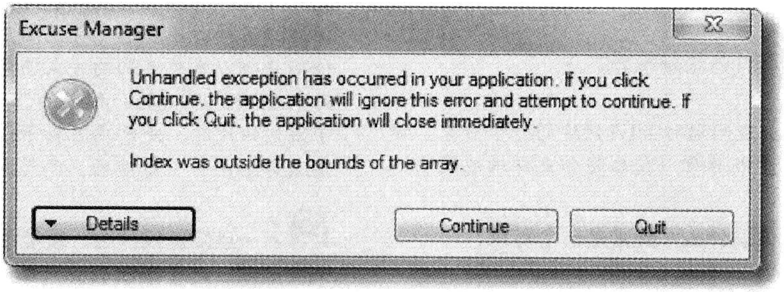
**答：** 不，很遗憾，有些情况下你的代码有问题，但是只通过查看代码很难找出问题出在哪里。正因如此，IDE提供了一个很有用的工具：调试工具(debugger)。利用调试工具，可以让程序暂停，逐语句地执行，在这个过程中检查各个变量和字段的值。因此要找出代码为什么与你预期的表现不同就会容易得多。可以利用这个最佳机会查找和修正异常，当然最好是从一开始就避免异常的发生。

**异常就是要帮助你找出并修正代码中未能达到预期的情况。**

# Brian的代码出了意外

编写Brian的借口管理系统时，没有想到用户会试图从一个空目录中抽取随机的借口。

❶ Brian将他的借口管理程序指向笔记本电脑上的一个空文件夹，并点击Random按钮，这就出问题了。下面来看能不能找出到底哪里出了问题。以下是他在IDE之外运行这个程序弹出的未处理异常窗口：




❷ 好的，这是一个很好的起点。这个消息告诉我们数组索引越界，对不对？所以下面来查找Random Excuse按钮事件处理程序代码中的数组：

```
private void randomExcuse_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

❸ 嗯？并没有数组。不过它使用了Excuse的一个重载构造函数来创建一个新Excuse对象。也许构造函数代码中有一个数组：

```
public Excuse(Random random, string Folder) {
    string[] fileNames = Directory.GetFiles(Folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

找到了！确实有个数组。肯定是试图对这个数组使用一个越界的索引。

**4** 可以看出，如果指向一个不包含任何文件的目录，Directory.GetFiles()会返回一个空数组。注意了，这一点是可以检查出来的！只需要增加一个检查，在打开文件之前先确保目录非空，这样一来，莫名其妙的未处理异常窗口就会代之以一个消息框，清楚地指出有关的信息。

```
private void randomExcuse_Click(object sender, EventArgs e) {
    string[] fileNames = Directory.GetFiles(selectedFolder,"*.excuse");
    if (fileNames.Length == 0) {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    } else {
        if (CheckChanged() == true) {
            CurrentExcuse = new Excuse(random, Folder);
            UpdateForm(false);
        }
    }
}
```

在创建Excuse对象之前，检查文件夹中的借口文件，可以避免抛出异常，还可以弹出一个很有帮助的消息框。

噢，我知道了。异常不一定不好。有时它们可以指示bug，不过大多数情况下，异常只能告诉我出现了原先没预料到的问题。



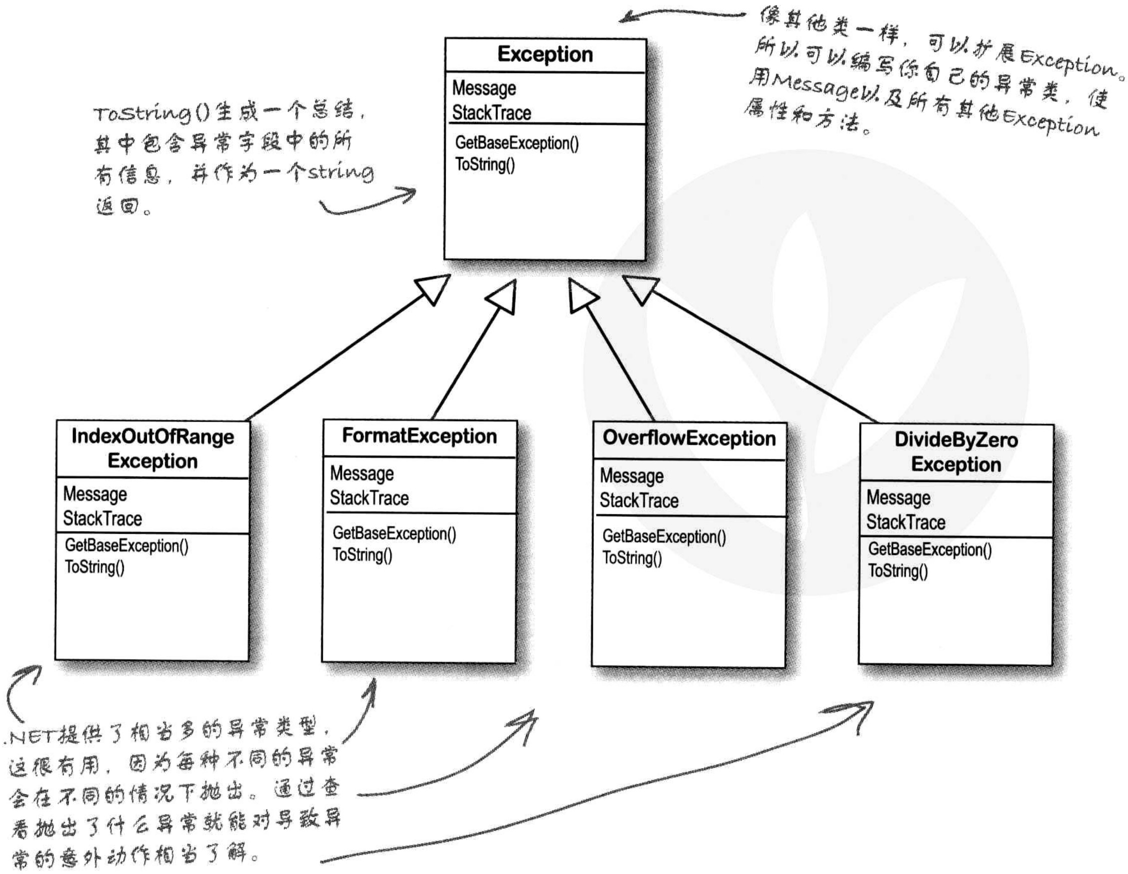
**没错。异常是一个很有用的工具，可以用来找出代码中哪些地方与你预期的表现不同。**

很多程序员第一次看到异常时都很头疼。不过异常实际上很有用，可以充分加以利用。看到一个异常时，它会给你提供很多线索，代码遇到原先没有预料到的情况时，这些线索可以帮助你找到问题。这对你来说实际上是一件好事：它能让你知道程序必须处理一种新情况，你可以利用这个机会做一些处理。

# 所有异常对象都继承自Exception

.NET提供了大量不同的异常类来表示可能需要报告的异常情况。由于其中很多异常都有类似的特性，所以这里可以充分利用继承。.NET定义了一个基类，名为Exception，所有特定的异常类型都继承自这个基类。

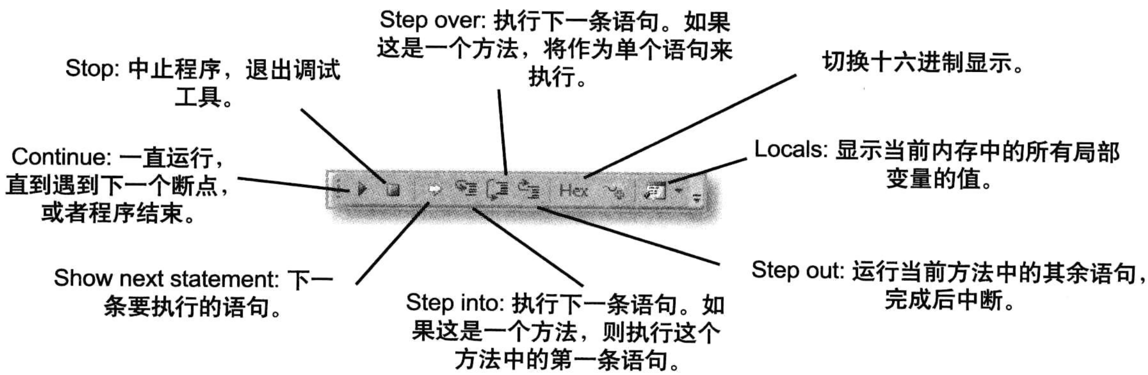
Exception类有一些很有用的成员。Message属性存储一个可读的消息，描述哪里出了问题。StackTrace可以告诉你出现这个异常时在执行什么代码，以及是什么导致了这个异常（当然还有其他一些成员，不过我们将先用到这两个属性）。



# 调试工具可以帮助跟踪和避免代码中的异常

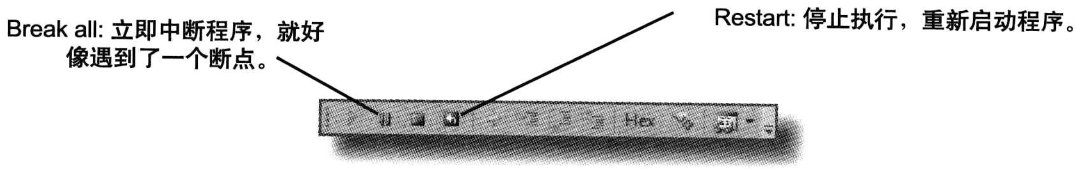
在程序中增加异常处理之前，需要知道程序中哪些语句会抛出异常。在这方面，IDE中内置的调试工具非常有用。完成本书中的例子里实际上你一直都在使用调试工具，不过现在花几分钟运行调试工具时，IDE会弹出一个工具条，其中包含一些很有用的按钮。可以试试看，把鼠标停在各个按钮上，看看它们能做什么。

只有在IDE中调试程序时才会显示调试工具条。所以，要想把鼠标停在这个工具条的按钮上，必须先运行一个程序。



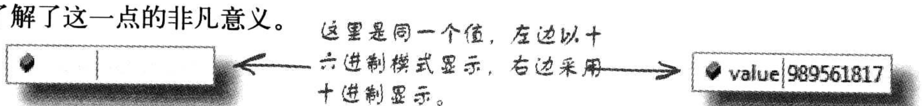

## 将IDE置为专家模式，展开Debug工具条

第一次使用Visual Studio 2010 Express时，会设置为基本设置（Basic Settings）模式，对于刚开始来讲这很好。不过，现在你已经使用了一段时间，可以改变一下模式。从菜单选择“ Tools”→“ Settings”→“Expert Settings”（IDE可能需要花点时间调整设置）。现在再来看调试工具条。你会看到它增加了两个新按钮（其他版本中这两个按钮已经出现在工具条上）：




## 切换十六进制模式

按下Hex按钮可以打开十六进制模式，然后把鼠标停在任意字段或变量上。然后再按这个按钮，可以关闭十六进制模式。IDE会自动为你将值转换为十六进制，上一章已经了解了这一点的非凡意义。



# 使用IDE的调试工具准确找出借口管理系统中哪里出了问题

下面使用调试工具更仔细地查看借口管理系统遇到的问题。前几章你可能已经多次用过调试工具，不过现在我们要一步一步地具体使用，确保你不会遗漏任何细节。

**动手调试**

**1 为Random按钮增加一个断点。**

首先要有一个起点，选择一个空文件夹后，点击Random Excuse按钮时会出现异常。所以打开这个按钮的代码，点击这个方法第一行代码中的某个位置，并从Debug菜单选择"Toggle Breakpoint"（或者按下F9），然后运行程序。选择一个空文件夹，并点击Random按钮，使程序在断点处中断：

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)     fileNames.Length == 0  true
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                    "No excuse files found");
    }
    else
    {
        if (CheckChanged())
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
}
```

鼠标停在fileNames.Length属性上，会显示浮动的表达式窗口，然后点击"图钉"将它固定住，使它不会消失。

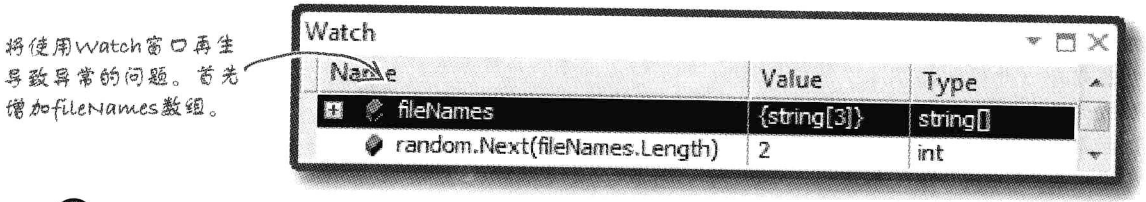
**2 单步跟踪事件处理程序，进入Excuse构造函数。**

使用Step Into命令（可以用工具条上的按钮，或者按下F11键），逐行地跟踪应用。由于选择了一个空的文件夹，所以应该会看到程序执行`MessageBox.Show()`，然后退出事件处理程序。

现在选择一个包含有借口的文件夹，再点击Random按钮，继续单步跟踪代码（一定要使用Step Into，而不是Step Over，不过你可能想一步跳过`CheckChanged()`方法）。到达创建新`Excuse`对象的那行代码时，会直接跳入构造函数内部。执行构造函数中的第一行代码，设置`fileNames`变量。然后把鼠标停在这个变量上查看它的值。

**3** **使用Watch窗口使问题再生。**

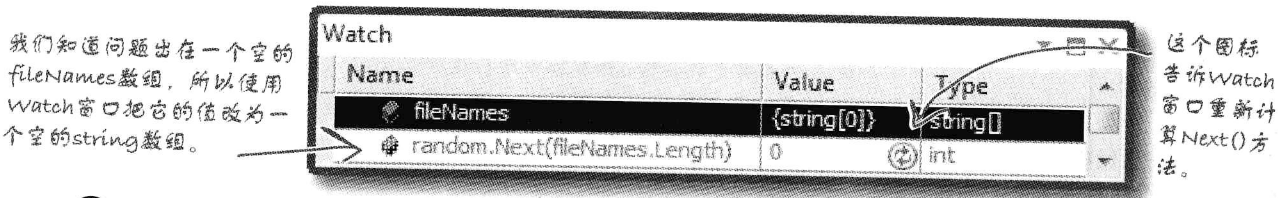
你已经了解Watch窗口的强大功能。现在使用它来再生异常。右键点击`fileNames`选择表达式“fileNames”，再选择“Add Watch”。它会在Watch窗口中增加一个监视项。然后点击fileNames下面的空行，输入表达式`random.Next(fileNames.Length)`告诉调试工具为它增加一个监视项。对于一个有3个借口的文件夹，Watch窗口将如下所示（所以`fileNames`长度为3）。




**4** **设置fileNames等于一个空的string数组。**

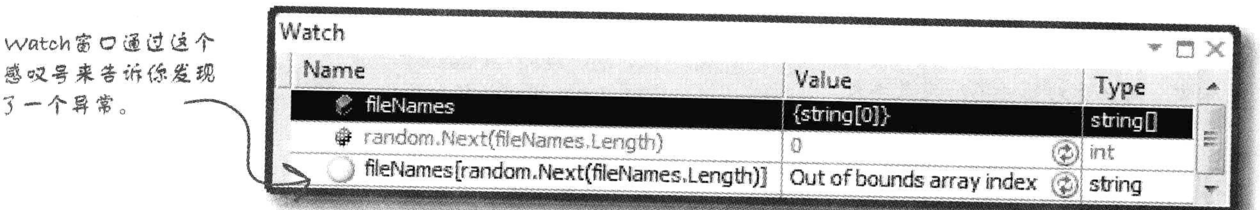
Watch窗口还有一个非常有用的特性，利用Watch窗口，可以修改所显示的变量和字段的值，甚至可以执行方法和创建新对象。如果使用了这个特性，它会显示一个重计算图标（），点击这个图标，通知它重新执行这一行，因为有时将同一个方法运行两次会生成不同的结果（如Random）。

双击`fileNames`的值—你会看到文本`{string[3]}`高亮显示。把它替换为**`new string[0]`**，就会立即看到发生两件事。首先，你会看到`fileNames`变量旁边的展开图标消失了，因为现在它为空。其次，`random.Next()`行会变灰，并有一个重计算图标()。点击这个图标再次执行方法，这应当返回0。




**5** **重新产生导致抛出Brian原异常的问题。**

现在调试有点意思了。向调试工具再增加一行代码，就是抛出异常的那行代码：`fileNames[random.Next(fileNames.Length)]`。一旦键入，Watch窗口就会执行这行代码……然后抛出异常。它会显示一个感叹号指出发现了异常，并在Value列中显示异常的文本。




**如果得到一个异常，可以再在调试工具中再生这个问题。这也是一种有效的方法，可以利用更具描述性的异常消息帮助你修正代码。**

## there are no Dumb Questions

**问：为什么Brian的未处理异常窗口看上去与IDE中的不一样？**

**答：** 这是因为，在IDE中运行程序就是在调试工具里运行，一旦截获到异常将立即中断程序（就像你按下了Break All按钮或者插入了断点一样），并在一个有用的窗口中显示这个异常。这样一来，就可以检查Exception对象以及程序中的字段和变量，从而找出问题。

Brian运行他的程序时，并没有从IDE运行。他发布了这个程序并进行了安装，就像你在第1章安装Contact List程序一样。也可以在IDE之外运行程序而无需发布，只要先构建程序，这样Visual Studio会创建一个可执行文件。查看你的工程下的bin/文件夹，它的某个子目录中应该有这个应用的exe可执行文件。运行这个文件，它抛出的所有异常都是未处理异常，这样就会显示Brian看到的窗口。

**问：这是怎么回事？如果在IDE之外运行程序，而且发生了一个异常，那么程序只能停下来，而我什么也不能做吗？**

**答：** 可以这么说，如果有一个未处理异常，你的程序确实会停止。但是这并不是说所有异常都是未处理异常！我们还会详细地介绍如何在代码中处理异常。用户看到的并不一定是未处理异常。

**问：我怎么知道把断点放在哪里呢？**

**答：** 这个问题问得好，对于这个问题并没有唯一的答案。代码抛出一个异常时，可以从抛出这个异常的语句开始设置断点，这是一种不错的想法。不过，一般来讲，实际上问题可能在较早前就已经发生了，这个异常只是前面那个问题的后续结果。例如，假设一条语句抛出了一个除0错误，这个语句中以某个值作为除数，而这个值是在10条语句之前生成的，只不过之前没有使用而已。所以把断点放在哪里并没有明确的答案，因为情况千差万别。不过，只要你很清楚你的代码是如何工作的，应该能找到一个好的起点。

**问：能在Watch窗口里运行方法吗？**

**答：** 当然可以。程序中任何合法的语句都可以在Watch窗口里运行，即使有些代码在Watch窗口中运行没有任何意义，但确实允许在Watch窗口中运行这些代码。下面给出一个例子。打开一个程序，启动运行，再中断，然后在Watch窗口中增加以下语句：System.Threading.Thread.Sleep(2000)（记住，这个方法会让程序延迟2秒）。现实中你不太可能这么做，但是看看这样做会有什么结果还是很有意思的：执行这个方法时会出现一个沙漏光标，它会持续2秒。之后，由于Sleep()没有返回值，Watch窗口会显示这样一个值："Expression has been evaluated and has no value"（表达式已计算但没有返回值），告诉你它没有返回任何结果，但是确实执行了这个方法。不仅如此，它还会显示智能提示窗口来帮助你向Watch窗口中键入代码。这很有用，因为即使程序正在运行，它也能告诉你某个对象有哪些方法。

**问：等一下，那么是不是说不能在Watch窗口中执行可能改变程序运行的代码？**

**答：** 完全可以运行这样的代码！虽然不是永久性的，但确实可以影响程序的输出。而且更有甚者，只是在调试工具中将鼠标停在字段上，就可能导致程序改变行为，因为鼠标停在一个属性上时，会执行它的获取存取方法。如果一个属性的获取存取方法要执行某个方法，那么鼠标停在这个属性上时就会执行该方法。而且如果这个方法设置了程序中的一个值，再次运行程序时，就会使用所设置的这个值，而这可能导致调试工具中出现一些不可预料的结果。程序员对这些看上去无法预料的随机结果取了一个名字：他们称之为heisenbug，这个开玩笑的说法起源于物理学家和他们做实验用的箱子里的猫。

**在IDE中运行程序时，未处理异常会导致程序中断，就好像遇到了断点一样。**

# 唉呀，代码还是有问题……

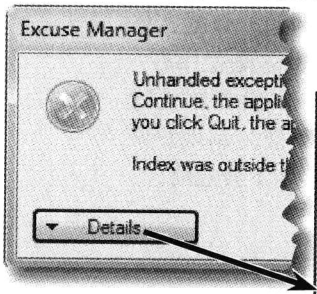





Brian很喜欢用他的借口管理系统，他记得第一次构建这个程序时有一个包含很多借口的文件夹，但是他忘了这个文件夹是在为程序增加串行化之前创建的。看看会发生什么…….

1. 你可以重新产生Brian的问题，只需使用记事本创建基于文本的借口文件。第一行应当是借口的描述，第二行是这个借口的效果，第三行应当是最后一次使用日期(“10/4/2007 12:08:13 PM”)。

2. 打开借口管理系统，再打开这个借口。此时会抛出一个异常！但是这一次可以点击Details按钮，更仔细地看看这个异常到底说了些什么。注意调用栈（call stack），一个方法被另一个方法调用，而后者又被另外一个方法调用，依此类推，这就称为调用栈（call stack）。




程序抛出了一个SerializationException。你能根据异常的详细内容找出是哪一行抛出了这个异常吗？

************** Exception Text **************

System.Runtime.Serialization.**SerializationException**: End of Stream encountered before parsing was completed.

at System.Runtime.Serialization.Formatters.Binary.__BinaryParser.Run()

at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)

at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage methodCallMessage)

at System.Runtime.Serialization.Formatters.Binary.**BinaryFormatter.Deserialize**(Stream serializationStream)

at Chapter10.**Excuse.OpenFile(String ExcusePath)** in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Excuse.cs:line 40

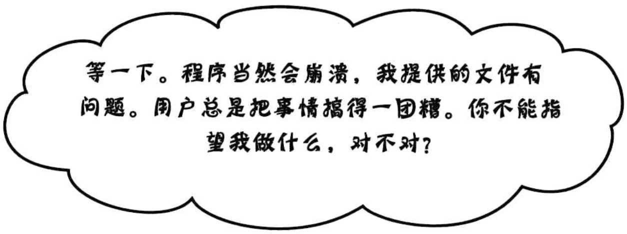
at Chapter10.**Excuse..ctor(Random random, String Folder)** in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Excuse.cs:line 30

at Chapter10.**Form1.randomExcuse_Click(Object sender, EventArgs e)** in C:\Documents and Settings\Administrator\My Documents\Visual Studio 2005\Projects\Chapter10\Chapter10\Form1.cs:line 146

看上去BinaryFormatter有问题，这是有道理的，因为它试图逆串行化一个文本文件。

从调用栈可以了解很多东西，它会告诉你哪些方法在运行。可以看到，Excuse类的OpenFile()方法由其构造函数（“.ctor”）调用，而这个构造函数又由“Random Excuse”按钮的点击事件处理程序调用。

3. 所以未处理异常窗口中的Details按钮会告诉你导致这个问题的很多有关情况。想想看你能对它做些什么？






**事实上，有些事情是你能做的。**

你说的没错，用户总是把事情搞糟，这是事实。但是这并不表示你什么也不能做。有些程序能妥善地处理有问题的数据、格式不正确的输入，以及其他没有预料到的情况，这些程序有一个名字：健壮程序（robust），而且C#提供了一些非常强大的异常处理工具来帮助你使程序更加健壮。尽管不能控制用户做什么，但是可以确保用户做了不该做的事情时程序不会崩溃。

> **ro-bust，形容词。**
> 结实的；能忍受或克服不利条件。经历过Tacoma Narrows大桥的灾难，土木工程小组为取代它的新桥寻找到一个更健壮的设计。

**Watch it!**

**如果串行化文件有问题，BinaryFormatter就会抛出一个异常。**

借口管理系统很容易抛出一个Serialization-Exception异常，只要输入的文件不是串行化的Excuse对象，就会导致这样的后果。由一个文件逆串行化一个对象时，BinaryFormatte希望这个文件包含一个串行化对象，而且与它要读取的类匹配。如果文件中包含了其他内容，不论是什么，Deserialize()方法都会抛出一个SerializationException异常。

# 用try和catch处理异常

在C#中，可以这么说，“尝试（try）运行这个代码，如果出现异常，用这些代码来捕获（catch）”。所尝试的代码是 try块，而处理异常的部分称为catch块。在catch块中，可以做一些事情，比如打印一条友好的错误消息，而不是让程序戛然而止：

```
private void randomExcuse_Click(object sender, EventArgs e)
{
  // ... code you added a few pages ago goes here ...
  try {
    if (CheckChanged() == true) {
      currentExcuse = new Excuse(random, selectedFolder);
      UpdateForm(false);
    }
  }
  catch (SerializationException) {
    MessageBox.Show(
      "Your excuse file was invalid.",
      "Unable to open a random excuse");
  }
}
```

这部分是try块。用try开始异常处理。在这里，我们把现有的代码放在try块中。

把可能抛出异常的代码放在try块中。如果没有发生异常，它会正常地运行，catch块中的语句将被忽略。但是，如果try块中的某个语句确实抛出了一个异常，就不会再执行try块中的其余语句。

catch关键字表示后面的块包含一个异常处理程序（exception handler）。

抛出一个异常时，程序会立即跳至catch语句，开始执行catch块。

这是最简单的一种异常处理：停止程序运行，写出异常消息，然后继续运行。

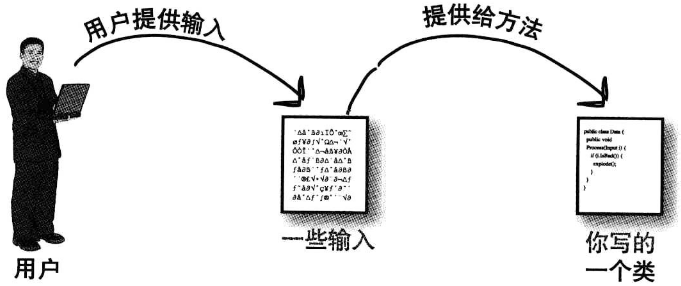
**BRAIN POWER**

既然抛出一个异常会导致代码自动跳至cathc块，那么异常发生之前处理的对象和数据会怎么样呢？

# 想调用的方法有风险会导致什么结果？

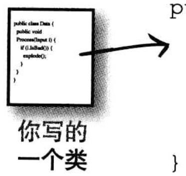
用户是难以预料的。他们可能会向你的程序输入各种各样奇怪的数据，而且可能会以你根本想不到的方式随意点击。没有关系，因为你可以利用妥善的异常处理来处理这些预料之外的输入。

**① 假设你的用户正在使用你的代码，并提供了一些预料之外的输入。**




**② 这个方法会做一些有风险的事情，在运行时可能无法正常执行。**

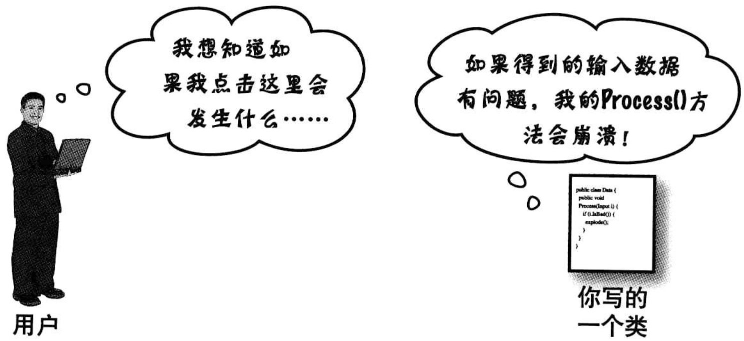
"运行时"（runtime）就是指"当你的程序正在运行时"。有些人把异常就称为"运行时错误"。



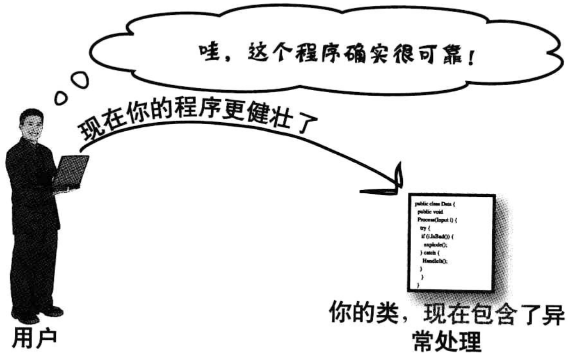

```
public void
   Process(Input i) {
   if (i.IsBad()) {
      explode();
   }
}
```

**③ 你要知道你调用的方法是有风险的。**

如果能想个办法做不那么有风险的事情，而避免抛出异常，这才是最好的做法！不过有些风险是无法避免的，这种情况下你可能就会希望这样做。




**④ 然后编写代码，即使出现错误也能够妥善处理。要事前做好准备，以防万一。**



## there are no Dumb Questions

**问：** **那么什么时候使用try和catch呢？**

**答：** 只要是编写有风险的代码，或者可能抛出异常的代码，都应该使用try和catch。关键是要明确哪些代码是有风险的，哪些代码比较安全。

你已经看到了，如果代码要使用用户提供的输入，就可能有风险。用户有可能提供了不正确的文件，提供了单词而不是你想要的数字，或者提供了名字而不是本该提供的日期，而且他们有可能在所能想象到的任何地方随意点击鼠标。作为一个好程序，必须能接受所有这些输入，平静地加以处理。也许不能为用户提供可以使用的结果，但是起码能让用户知道它发现了问题，还可以对如何解决问题提供建议。

**问：** **既然程序无法提前知道问题，又怎么提供解决问题的建议呢？**

**答：** 这正是catch块的作用。只有当try块中的代码抛出一个异常时才会执行catch块。你要利用这个机会让用户知道出问题了，并告诉用户这种情况是可以修正的。

输入不正确时，如果借口管理系统只是简单地崩溃，那么这个程序实在没多大用处。但是，如果它试图读输入，并在窗体上显示一堆“垃圾”，那也没有意义—实际上，有些人可能会说这样更糟糕。不过，倘若程序能显示一个错误消息，告诉用户无法读这个文件，用户就能知道出了问题，而且能得到有关的一些信息，利用这些信息能够处理这个问题。

**问：** **这么说调试工具只用来排除异常吗？**

**答：** 不。在这本书中你已经多次看到，调试工具实际上是一个非常有用的工具，可以用来检查你写的任何代码。有时，可以用调试工具单步跟踪你的代码，检查某些字段和变量的值，比如说，如果有一个非常复杂的方法，你可能希望确保它能正确地工作。

不过另一方面，从“调试工具”这个名字也可以看出，它最常用的用途就是跟踪和消除bug。有时，这些bug就是抛出的异常。但是大多数情况下，还可以使用调试工具查找其他类型的问题，例如代码没有给出你期望的结果时，就可以利用调试工具来查找原因。

**问：** **我还是不完全了解Watch窗口。Watch窗口到底用来做什么？**

**答：** 调试一个程序时，往往希望关注某些变量和字段如何变化。这就引入了Watch窗口。如果在Watch窗口中增加了一些变量来进行监视，每次跟踪代码（不论采用step into、step out还是step over）时Watch窗口都会更新它们的值。这样一来，你就能监视执行每条语句后它们发生了什么变化，如果你想查出某个问题，这会非常有用。

Watch窗口还允许你键入任何语句，它会执行你键入的语句。如果这个语句更新了程序中的某个些字段和变量，Watch窗口中也会更新这些值。这样一来，即使程序正在运行，也能更改值，这也是一个很有用的工具，可以实现异常和其他bug的再生。

Watch窗口中完成的所有修改只会影响内存中的数据，而且只持续到程序运行结束。重启程序时，在Watch窗口中修改的值又会还原。

**只有try块中的代码抛出异常时才会执行catch块。可以利用这个机会向用户提供有关信息来修正这个问题。**

# 使用调试工具完成try/catch流程

异常处理中有很重要的一点，try块中的一个语句抛出一个异常时，这个块中的其余代码会被“短路”，即不再执行。程序的执行会立即跳至catch块中的第一行代码。不过不要只听我们的一面之词，自己试试看……

**动手调试**

❶ 把将这一章的所有代码放在借口管理系统中Random Excuse按钮的Click事件处理程序中。在事件处理程序的第一行上放一个断点。然后在IDE中运行程序。单击Folder按钮，指定一个只包含一个借口文件的文件夹，确保这个文件并不是一个合法的借口文件（但确实有“.excuse”扩展名）。点击Random Excuse按钮。调试工具会在先前旋转断点的位置中断程序。点击6次“Step Over”按钮（或按下6次F10），执行到调用Excuse构造函数的语句。现在调试工具的屏幕应该如下所示：

这是之前在事件处理程序第一行放置的断点。

用Step over方式单步执行语句，直到黄色的“下一条语句”条显示要执行的下一条语句将创建新的Excuse对象。

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                "No excuse files found");
    }
    try
    {
        if (CheckChanged() == true)
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
    catch (SerializationException)
    {
        MessageBox.Show(
          "Your excuse file was invalid.",
          "Unable to open a random excuse");
    }
}
```

在调试工具中要确保使用Step Over (F10) 命令，这样就不会单步跟踪进入到CheckChanged()方法内部。

❷ 使用Step Into (F11)单步跟踪进入这条new语句。调试工具将跳至Excuse构造函数，把黄色的“下一条语句”条放在声明代码行上。继续使用Step Into (F11)单步跟踪进入OpenFile()方法。观察到达Deserialize()行时会发生什么。

一旦单步进入到创建Excuse对象的new语句，调试工具就会跳到构造函数代码中。

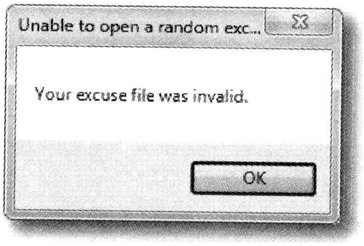
```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

**3** 继续跟踪代码。调试工具一旦执行`Deserialize()`语句，就会抛出异常，程序会立即短路，跳过`UpdateForm()`调用不再执行，而直接跳到`catch`块。

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                    "No excuse files found");
    }
    try
    {
        if (CheckChanged() == true)
        {
            currentExcuse = new Excuse(random, selectedFolder);
            UpdateForm(false);
        }
    }
    catch (SerializationException)
    {
        MessageBox.Show(
          "Your excuse file was invalid.",
          "Unable to open a random excuse");
    }
}
```

调试工具会用黄色的“下一条语句”条突出显示catch语句，但这个块中的其余语句都显示为灰色，表示这些都要执行。

**4** 按下Continue按钮（或F5）再启动程序。它会从黄色“下一条语句”条突出显示的位置重新开始运行程序，在这里就是从catch块继续运行。




**Watch it!**

**当心构造函数中的异常！**

现在你可能已经注意到了，构造函数没有返回值，甚至连void都没有。这是因为，构造函数并不返回任何结果。它的唯一作用就是初始化一个对象，这对于构造函数中的异常处理就带来了一个问题。在构造函数中抛出一个异常时，试图实例化这个类的语句不会得到这个对象的实例。出于这个原因，所以必须把try/catch块移到按钮的事件处理程序中。这样一来，如果构造函数中出现一个异常，代码不必指望CurrentExcuse包含一个合法的Excuse对象。

这个提示对你的职业发展很有意义：很多C#编程职位面试时都有这样的问题，考察你如何处理构造函数中的异常。

# 如果有些代码总要运行，可以使用一个finally块

程序抛出一个异常时，可能发生几种情况。如果异常没有得到处理，程序会停止运行而崩溃。如果异常得到了处理，代码会跳至catch块。但是，try块中的其余代码会怎么样呢？如果你要关闭一个流，或者要清理一些重要的资源呢？这些代码必须运行，即使出现了异常也需要运行这些清理代码，否则程序的状态就会乱七八糟。这里finally块就能派上用场了。它放在try和catch块后面。**finally**块总会运行，而不论是否抛出异常。可以使用**finally**块结束Random Excuse按钮的事件处理，如下所示：

```
private void randomExcuse_Click(object sender, EventArgs e) {
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0) {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    } else {
        try {
            if (CheckChanged() == true) {
                currentExcuse = new Excuse(random, selectedFolder);
            }
        }
        catch (SerializationException) {
            currentExcuse = new Excuse();
            currentExcuse.Description = "";
            currentExcuse.Results = "";
            currentExcuse.LastUsed = DateTime.Now;
            MessageBox.Show(
              "Your excuse file was invalid.",
              "Unable to open a random excuse");
        }
        finally {
            UpdateForm(false);
        }
    }
}
```

finally块确保不论是否抛出一个异常都会运行UpdateForm()。所以，如果Excuse构造函数成功地读取了一个借口，它会调用UpdateForm()，不过如果构造函数抛出一个异常并清空了借口，也同样会调用这个方法。

如果Excuse构造函数抛出一个异常，我们无法知道CurrentExcuse有什么。不过，可以明确的是，不会创建Excuse的任何实例。所以catch块创建了一个新的Excuse对象，并清空它的所有字段。

> **SerializationException** 在System.Runtime.Serialization命名空间中，所以需要在窗体文件最前面增加**using System.Runtime.Serialization;**。

要捕获类似SerializationException的特定的异常。catch语句的后面通常要有一种特定的异常，指出要捕获什么。如果只有“catch (Exception)”也是一个合法的C#代码，甚至可以省略异常类型，只使用**catch**。如果是这样，就会捕获所有异常，而不论抛出什么类型的异常。不过建立像这样的“全能型”异常处理程序是一种非常不好的做法。你的代码要捕获尽可能特定的异常。

## 继续调试

❶ 用前一页上的代码更新Random Excuse按钮的事件处理程序。然后在这个方法中第一行上放一个断点，调试程序。

❷ 正常地运行程序，确保将程序的文件夹设置为某个包含大量正常借口文件的文件夹时，Random Excuse按钮的工作正常。调试工具会在你设置的断点处中断：

"下一条语句"条和断点在同一行上，IDE会在边缘的红点上显示一个黄色箭头。

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    string[] fileNames = Directory.GetFiles(selectedFolder, "*.excuse");
    if (fileNames.Length == 0)
    {
        MessageBox.Show("Please specify a folder with excuse files in it",
                        "No excuse files found");
    }
    else
    {
        try
        {
            if (CheckChanged() == true)
            {
                currentExcuse = new Excuse(random, selectedFolder);
            }
        }
        catch (SerializationException)
        {
            currentExcuse = new Excuse();
            currentExcuse.Description = "";
            currentExcuse.Results = "";
            currentExcuse.LastUsed = DateTime.Now;
            MessageBox.Show(
              "Your excuse file was invalid.",
              "Unable to open a random excuse");
        }
        finally
        {
            UpdateForm(false);
        }
    }
}
```

❸ 单步执行Random Excuse按钮事件处理程序中的其余代码，保证按预期的方式运行。它应当完成try块，跳过catch块（因为没有抛出任何异常），然后执行finally块。

❹ 现在再来设置程序的文件夹，使它指向的文件夹中只包含一个格式不正确的借口文件，再点击Random Excuse按钮。这会执行try块，然后当抛出异常时跳至catch块。完成catch块中的所有语句后，将执行finally块。

## there are no Dumb Questions

**问：等一下。这么说，每次程序遇到一个异常时，都会停止它正在做的工作，除非我专门编写了代码来捕获这个异常。这怎么能算是好事呢？**

**答：** 异常最大的好处之一就是，你能清楚地认识到遇到了问题。可以想象一下，在一个复杂的应用中，很容易对程序处理的所有对象失去控制。异常可以提醒你注意存在的问题，并帮助你追寻出问题的根由，使你相信程序在做你希望它做的事情。

程序中出现异常时，你本来希望发生的一些事情没有发生。也许是一个对象引用没有指向它本来应该指向的对象，或者有可能用户提供了你没有想到的一个值，也可能你要处理的文件突然没有了。如果发生了诸如此类的问题，而你不知道，程序很有可能会有不正确的输出，而且从这个问题开始，程序的行为可能与你原先写程序时的期望完全不同。

现在假设你不知道出现了错误，你的用户开始打电话告诉你数据不正确，程序不稳定。正是因为这个原因，我们说异常能中断程序正在做的工作是一件好事。异常要求你在容易发现和修正问题时及时地进行处理。

**问：既然如此，那么什么是已处理异常，什么是未处理异常呢？**

**答：** 只要程序抛出一个异常，运行时环境就会搜索你的代码，查找与这个异常匹配的catch块。如果你编写了这样一个catch块，就会执行这个catch块，并完成你为这个特定异常指定的处理。由于你提前写了一个catch块来处理这个错误，所以这个异常被认为是已处理异常（handled exception）。如果运行时环境没有找到与这个异常匹配的catch块，就会停止程序正在做的工作，而产生一个错误。所以，这个异常被称为未处理异常（unhandled exception）。

**问：使用一个“全能”异常不是更容易吗？编写能捕获每一个异常的代码不是更安全吗？**

**答：** 要尽量避免捕获Exception，而应当捕获特定的异常。你听过这样一句老话吧？防患于未然才是上策。在异常处理方面尤其如此。依赖于这种这种全能异常的做法往往只是对不良编程的一种补救。例如，在打开一个文件之前最好使用Exists()检查文件是否存在，而不是捕获一个FileNotFoundException。尽管有些异常是无法避免的，但你会发现其中很大一部分原本根本不必抛出。有时保持异常未处理确实很有用。真实世界的程序往往有很复杂的逻辑，通常很难在出问题时正确地恢复，特别是程序中埋藏很深的问题。只需处理特定的异常，避免这种“全能型”异常处理程序，让异常出现时能够在顶层得到捕获，就能得到更健壮的代码。

**问：如果catch没有指定特定的异常会怎么样呢？**

**答：** 这样的catch块会捕获try块中可能抛出的任何类型的异常。

**问：既然没有指定特定异常的catch块会捕获所有异常，那又何必指定异常呢？**

**答：** 这个问题问得好。这是因为，对不同异常可能需要做不同的处理才能保证程序继续运行。对于除0产生的异常，相应的catch块可能要设置某些数字值，从而保存你已经处理的一些数据。对于null引用异常，如果想从这个异常恢复，则可能需要创建一个对象的新实例。

**问：是不是所有错误处理都按try/catch/finally顺序来完成？**

**答：** 不是的。可以有另外的组合。如果想处理很多不同类型的错误，就可以有多个catch块。也可以根本没有catch块。try/finally块是完全合法的。它不会处理任何异常，但是能确保即使try块执行到一半时中止，finally块中的代码也总能运行。不过稍后还会详细讨论这种情况……

**出现未处理异常说明程序会以不可预知的方式运行。正因如此，只要遇到未处理异常，程序就会中止。**

# 池塘谜题



你的任务是从池塘里取出代码片段，在程序中填空。同一个片段可以使用多次，而且并不是所有片段都要用到。你的目标是建立一个能生成以下输出的程序。

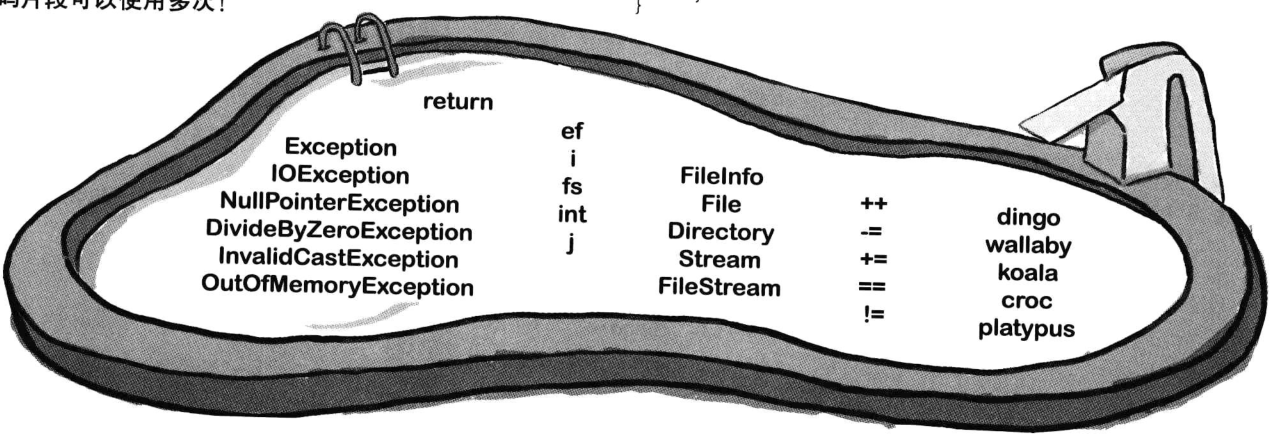
**输出：** ⟶ **G'day Mate!**

```
using System.IO;
public static void Main() {
    Kangaroo joey = new Kangaroo();
    int koala = joey.Wombat(
          joey.Wombat(joey.Wombat(1)));
    try {
        Console.WriteLine((15 / koala)
                + " eggs per pound");
    }
    catch (___________________) {
        Console.WriteLine("G'Day Mate!");
    }
}
```

```
class Kangaroo {
  ___________ fs;
  int croc;
  int dingo = 0;

  public int Wombat(int wallaby) {
    _______ __;
    try {
      if (________ > 0) {
        __ = _____.OpenWrite("wobbiegong");
        croc = 0;
      } else if (________ < 0) {
        croc = 3;
      } else {
        ___ = _____.OpenRead("wobbiegong");
        croc = 1;
      }
    }
    catch (IOException) {
      croc = -3;
    }
    catch {
      croc = 4;
    }
    finally {
      if (______ > 2) {
        croc ___ dingo;
      }
    }
    ________ ______;
  }
}
```

**注意：池塘里的每个代码片段可以使用多次！**




池塘谜题越来越难了，这些名字也越来越含糊，尽量减少给你的提示。这个问题确实需要好好想想！要记住，谜题不是非完成不可的，所以如果需要先看其他内容以后再回来解决这个问题也完全可以……不过，如果你真的希望把这个内容记在大脑里，谜题会很有帮助！

# 池塘谜题答案



Joey.Wombat()调用了3次，第3次返回一个0。这会导致WriteLine()抛出一个DivideByZeroException。

```
public static void Main() {
    Kangaroo joey = new Kangaroo();
    int koala = joey.Wombat(joey.Wombat(joey.Wombat(1)));
    try {
        Console.WriteLine((15 / koala) + " eggs per pound");
    }
    catch (DivideByZeroException) {
        Console.WriteLine("G'Day Mate!");
    }
}
```

这个catch块只捕获代码的除0异常。

这是一个FileStream，线索是它有一个OpenRead()方法，而且抛出一个IOException。

```
class Kangaroo {
    FileStream fs;
    int croc;
    int dingo = 0;

    public int Wombat(int wallaby) {
        dingo ++;
        try {
            if (wallaby > 0) {
                fs = File.OpenWrite("wobbiegong");
                croc = 0;
            } else if (wallaby < 0) {
                croc = 3;
            } else {
                fs = File.OpenRead("wobbiegong");
                croc = 1;
            }
        }
        catch (IOException) {
            croc = -3;
        }
        catch {
            croc = 4;
        }
        finally {
            if (dingo > 2) {
                croc -= dingo;
            }
        }
        return croc;
    }
}
```

这个代码打开一个名为“wobbiegong”的文件，并且第一次调用时使这个文件一直打开。然后再次打开这个文件。不过由于它之前没有关闭文件，这会导致抛出一个IOException。

你已经知道了，处理完文件后一定要把它们关闭。如果没有做到，文件会锁定为打开状态，如果试图再次打开，就会抛出一个^IOException。

要记住，代码中应当避免全能型异常。不过要知道为了让这个谜题更有意思，我们做了一些处理，你要避免这些做法，比如说使用这种含糊的变量名。

# 使用Exception对象得到问题的有关信息

我们一直在说，抛出一个异常时.NET会生成一个Exception对象。编写catch块时，可以访问这个对象。请看它是如何工作的：

**1** 一个对象正在完成它的工作，遇到一些未预料到的情况时，它抛出一个异常。

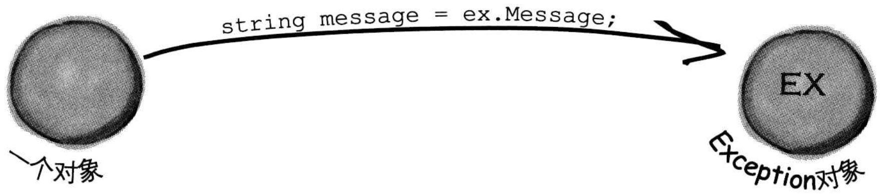



**2** 好在，它的try/catch块捕获了这个异常。在catch块中，它为这个Exception指定了一个名字：ex。

在catch块中指定一个特定类型的异常时，如果提供了一个变量名，代码就可以使用这个变量名来访问这个Exception对象。

```
try {
    DoSomethingRisky();
}
catch (RiskyThingException ex) {
    string message = ex.Message;
    MessageBox.Show(message, "I took too many risks!");
}
```

**3** Exception对象一直存在，直到catch块结束处理。然后**ex**引用消失，这个对象最终被回收。



# 使用多个catch块处理多种类型的异常

你已经知道，可以捕获一种特定类型的异常……，但是如果编写的代码中可能出现多种问题呢？在这些情况下，可能希望编写代码来处理每一种不同类型的异常。此时就可以使用多个catch块。下面是一个例子，这里的代码取自蜂巢花露处理工厂。可以看到，它捕获了多种类型的异常。其中一些情况下使用了Exception对象的属性。使用Message属性非常常见，这个属性中通常包含抛出的异常的一个描述。也可以调用throw;重新抛出这个消息，这样它就能在调用栈更高层得到处理。

还可以调用异常的ToString()方法来得到大量有关信息放入MessageBox中。

```
public void ProcessNectar(NectarVat vat, Bee worker, HiveLog log) {
    try {
        NectarUnit[] units = worker.EmptyVat(vat);
        for (int count = 0; count < worker.UnitsExpected, count++) {
            stream hiveLogFile = log.OpenLogFile();
            worker.AddLogEntry(hiveLogFile);
        }
    }
    catch (VatEmptyException) {
        vat.Emptied = true;
    }
    catch (HiveLogException ex) {
        throw;
    }
    catch (IOException ex) {
        worker.AlertQueen("An unspecified file error happened: "
            + "Message: " + ex.Message + "\r\n"
            + "Stack trace: " + ex.StackTrace + "\r\n"
            + "Data: " + ex.Data + "\r\n");
    }
    finally {
        vat.Seal();
        worker.FinishedJob();
    }
}
```

有时你可能希望使用throw;把一个异常向上发送到调用这个方法的另一个方法，这就是在重新抛出这个异常。

如果不打算使用Exception对象，就没有必要声明。

有多个catch块时，会按顺序检查。在这个代码中，首先会查看VatEmptyException异常，然后检查HiveLogException。最后一个catch块会捕获IOException，这是多种不同文件异常的基类，包括FileNotFoundException和EndOfStreamException。

这个catch块将异常赋至变量ex，可以使用这个变量得到这个Exception对象的有关信息。

两个catch块中Exception有相同的变量名（"ex"）是允许的。

这个语句使用了Exception对象中的3个属性：Message，其中包含IDE异常窗口中常见的消息（"Attempted to divide by zero"）；StackTrace，它会给出调用栈的一个总结；Data，有时这个属性中包含与这个异常有关的一些重要数据。

# 一个类抛出异常，另一个类捕获异常

当然，一个类中的一个方法也可以抛出异常，由同一个类中的另一个方法捕获。

构建一个类时，并不一定知道将来会怎样使用这个类。有时其他人使用这个类对象的方法可能不当，以至于带来问题，有时甚至你自己都会犯这种错误！因此引入了异常。

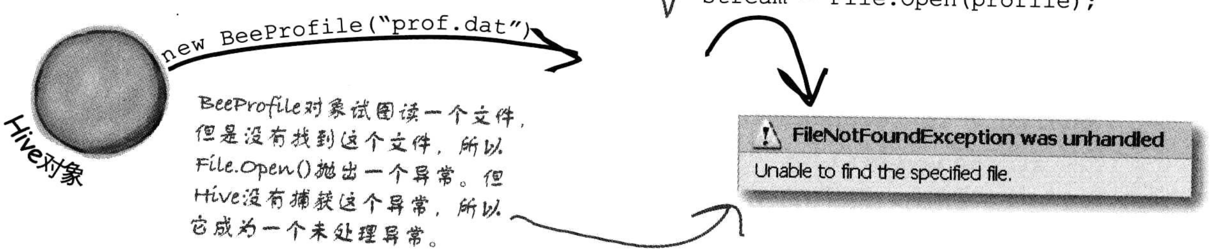
抛出异常的目的是查看哪里可能出问题，以便提前做好应急方案。并不总是在一个方法内抛出异常再捕获这个异常。异常往往在一个方法中抛出，而在另一个完全不同的方法中捕获，而且往往在一个完全不同的对象中。

**不要这样做……**

如果没有妥善的异常处理，一个异常就会中止整个程序。一个为蜂王管理个人简介的程序是这样做的：

这个BeeProfile对象的构造函数希望得到一个简介数据文件的文件名，然后可以使用File.Open()打开这个文件。如果打开文件时出现问题，程序就会异常中止。

```
stream = File.Open(profile);
```

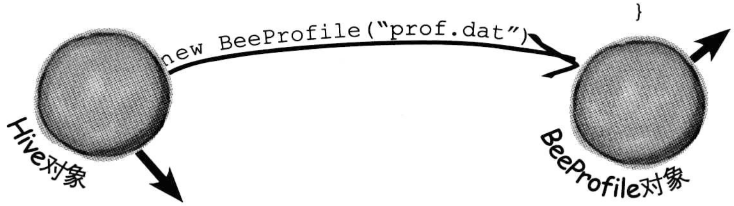



注意BeeProfile对象如何截获异常，并使用其WriteLogEntry()方法记入日志，然后再将异常抛出，传递到Hive。

```
try {
   stream = File.Open(profile);
} catch (FileNotFoundException ex) {
   WriteLogEntry("unable to find " +
      profile + ": " + ex.Message();
   throw;
}
```

**……可以这样做。**

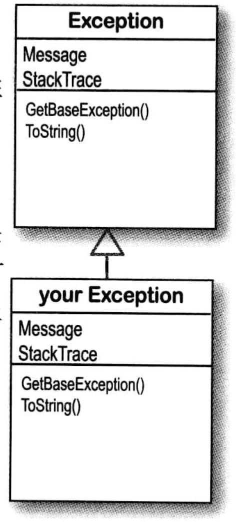
BeeProfile对象可以截获这个异常，并增加一个日志记录。然后反过来把异常抛回给Hive，由它捕获并妥善恢复。




```
try {
   prof = new BeeProfile("prof.dat");
} catch (FileNotFoundException) {
   Hive.RecreateBeeProfile("prof.dat");
}
```

现在Hive尝试创建一个新的BeeProfile对象，如果传入了一个非法的文件名，可以相信BeeProfile会记下这个错误，然后抛出一个异常来警告出现了问题。Hive可以捕获这个异常，并采取适当的动作，在这里就是重新创建蜜蜂的个人简介。

# 蜜蜂需要一个OutOfHoney异常




你的类还能抛出自己的异常。例如，如果本来需要一个值的方法得到了一个null参数，往往会抛出.NET方法抛出的同样的异常：

如果方法参数得到非法或意料外的值，就会抛出这个异常。

```
throw new ArgumentNullException();
```

但是有时你可能希望程序根据运行时发生的一种特殊情况抛出异常。例如，蜂巢里的蜜蜂会根据其重量以不同的速度消耗蜂蜜。如果蜂蜜用完了，就有必要让蜂巢抛出一个异常。可以创建一个定制异常来处理这种特定的错误条件，为此只需创建你自己的类，让它继承Exception，然后在遇到一个特定错误时抛出这个异常。

```
class OutOfHoneyException : System.Exception {
    public OutOfHoneyException(string message) : base(message) { }
}
class HoneyDeliverySystem {
    ...
    public void FeedHoneyToEggs() {
       if (honeyLevel == 0) {
            throw new OutOfHoneyException("The hive is out of honey.");
       } else {
            foreach (Egg egg in Eggs) {
               ...
}
public partial class Form1 : Form {
...
   private void consumeHoney_Click(object sender, EventArgs e) {
       HoneyDeliverySystem delivery = new HoneyDeliverySystem();
       try {
            delivery.FeedHoneyToEggs();
       }
       catch (OutOfHoneyException ex){
          MessageBox.Show(ex.Message, "Warning: Resetting Hive");
          Hive.Reset();
       }
   }
}
```

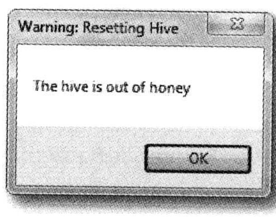
需要为你的异常创建一个类，并确保它继承了System.Exception。注意，我们重载了构造函数，从而传入一个异常消息。

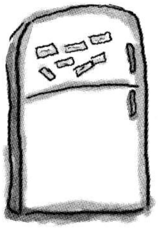
这会抛出异常对象的一个新实例。

如果蜂巢里还有蜂蜜，就不会抛出这个异常，代码会继续运行。

与其他异常一样，可以按名捕获一个定制异常，并根据需要进行处理。

在这里，如果蜂巢里的蜂蜜用完了，所有蜜蜂都不能工作，所以模拟系统无法继续运行。一旦蜂蜜用光，要让程序继续工作，惟一的办法就是重置蜂巢，为此可以在catch块中放上重置代码。





# 异常磁条

整理这些磁条，使应用向控制台写出以下输出。

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

输出:

**when it thaws it throws.**

```
}
```

```
}
```

```
}
```

```
if (t == "yes") {
```

```
Console.Write("a");
```

```
Console.Write("o");
```

```
Console.Write("t");
```

```
Console.Write("w");
```

```
Console.Write("s");
```

```
try {
```

```
} catch (MyException) {
```

```
throw new MyException();
```

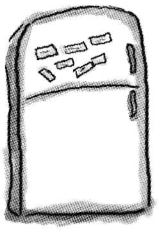
```
} finally {
```

```
doRisky(test);
```

```
        Console.Write("r");
    }
}
```

```
class ExTestDrive {
    public static void Zero(string test) {
```

```
static void DoRisky(String t) {
    Console.Write("h");
```



# 异常磁条答案

整理这些磁条，使应用向控制台写出以下输出。

```
public static void Main() {
    Console.Write("when it ");
    ExTestDrive.Zero("yes");
    Console.Write(" it ");
    ExTestDrive.Zero("no");
    Console.WriteLine(".");
}

class MyException : Exception { }
```

**输出:**

**when it thaws it throws.**

这一行定义一个名为MyException的定制异常，将在这个代码的一个catch块中捕获这个异常。

```
class ExTestDrive {
   public static void Zero(string test) {
      try {
         Console.Write("t");
         doRisky(test);
         Console.Write("o");
      } catch (MyException) {
         Console.Write("a");
      } finally {
         Console.Write("w");
      }
      Console.Write("s");
   }

   static void DoRisky(String t) {
      Console.Write("h");
      if (t == "yes") {
         throw new MyException();
      }
      Console.Write("r");
   }
}
```

Zero()方法会输出“thaws”或“throws”，这取决于向test参数传入“yes”还是传入其他值。

finally块确保这个方法总会输出“w”，而且由于“s”在异常处理程序外部输出，所以“s”也总会显示。

只有当doRisky()没有抛出异常时才会执行这一行。

doRisky()方法只在传入串“yes”时抛出一个异常。

## BULLET POINTS

- 如果运行时出现问题，任何语句都能抛出一个异常。
- 使用一个try/catch块来处理异常。未处理异常会导致程序中止执行，并弹出一个错误窗口。
- try语句后的代码块中如果出现异常，会导致程序的执行会立即跳至catch之后代码块中的第一条语句。
- Exception对象可以提供所捕获异常的有关信息。如果在catch语句中指定了一个Exception变量，这个变量会包含try块中所抛出异常的有关信息：

```
try {
  // statements that might
  // throw exceptions
} catch (IOException ex) {
  // if an exception is thrown,
  // ex has information about it
}
```

- 可以捕获多种不同类型的异常。每个异常有其自己的对象，均继承自Exception。要尽量避免只捕获Exception—而应当捕获特定的异常。
- 每个try可以有多个catch语句：

```
try { ... }
catch (NullReferenceException ex) {
   // these statements will run if a
   // NullReferenceException is thrown
}
catch (OverflowException ex) { ... }
catch (FileNotFoundException) { ... }
catch (ArgumentException) { ... }
```

- 你的代码可以使用throw抛出异常：

```
throw new Exception("Exception message");
```

- 你的代码还可以使用throw;重新抛出一个异常，不过这只适用于catch块内部。重新抛出异常会保证调用栈不变。
- 可以通过继承Exception基类创建定制异常。

```
class CustomException : Exception;
```

- 大多数情况下，只需要抛出.NET内置的异常，如ArgumentException。之所以使用不同类型的异常，是因为这样可以为用户提供更多信息。如果弹出一个窗口，其中只有文本“An unknown error has occurred”（出现一个未知错误），这样并没有多大的用处，但是如果给出一个错误消息，指出“The excuse folder is empty. Please select a different folder if you want to read excuses.”（借口文件夹为空。如果想读取借口请选择另外的文件夹），这会更有用。

## 避免大量问题的一种简便方法：利用using，可以轻松使用try和finally

要记住，在“using”语句中声明一个引用时，会在这个块结束时自动调用其Dispose()方法。

你已经知道了，using是保证文件妥善关闭的一种简便方法。可你不知道的是，using实际上是C#为try和finally提供的一个便捷方式！

```
using (YourClass c
        = new YourClass() ) {

   // code

}
```

等同于

```
YourClass c = new YourClass();

try {

    // code

} finally {

    c.Dispose();
}
```

使用using语句时，也就是充分利用finally来确保一定会调用其Dispose()方法。

# 避免异常：实现IDisposable完成自己的清理

流很棒，因为已经为它们编写了一些代码，能够在撤销对象时自行关闭。但是，如果你自己的定制对象撤销时总要做一些收尾工作，该怎么做呢？如果能写一些代码，使得在using语句中使用这个对象时能够运行你写的这些代码，那不是很棒吗？

IDisposable是避免常见异常和问题的一个非常有效的方法。如果要使用实现了这个接口的类，一定要使用using语句。

在C#中这是可以的，可以利用IDisposable接口做到。通过实现IDisposable接口，在Dispose()方法中编写你自己的清理代码，如下所示：

如果一个类实现了IDisposable，就只能在"using"语句中使用这个类；否则，程序将不能编译。

如果想在一个using语句中使用一个对象，这个对象必须实现IDisposable。

```
class Nectar : IDisposable {
  private double amount;
  private BeeHive hive;
  private Stream hiveLog;
  public Nectar(double amount, BeeHive hive, Stream hiveLog) {
    this.amount = amount;
    this.hive = hive;
    this.hiveLog = hiveLog;
  }
  public void Dispose() {
    if (amount > 0) {
       hive.Add(amount);
       hive.WriteLog(hiveLog, amount + " mg nectar added to the hive");
       amount = 0;
    }
  }
}
```

IDisposable接口只有一个成员：Dispose()方法。这个方法中的所有代码都会在using语句的最后执行……或者在手动调用Dispose()时执行。

编写这个Dispose()方法时考虑到要调用多次，而不只一次。

这部分代码会清空剩余的花蜜，放在蜂巢里，并记录一个消息。这很重要，而且必须完成，所以要把它放在Dispose()方法中。

**实现IDispose的一个原则是你的Dispose()方法可以调用多次而没有任何副作用。这是一个重要的原则，你能想出为什么吗？**

现在可以使用多个using语句。下面使用一个实现了IDisposable的内置对象：Stream。然后处理更新后的Nectar对象，它也实现了IDisposable接口：

需要在同一个代码块中声明两个IDisposable引用时就会看到这样的嵌套using语句。

```
using (Stream log = new File.Write("log.txt"))
using (Nectar nectar = new Nectar(16.3, hive, log)) {
      Bee.FlyTo(flower);
      Bee.Harvest(nectar);
      Bee.FlyTo(hive);
}
```

Nectar对象使用了Log流，这个流会在外层using语句的最后自动关闭。

然后Bee对象使用Nectar对象，Nectar对象会在内层using语句的最后自动把花蜜增加到蜂巢。

## there are no Dumb Questions

**问：在using语句中只能使用实现了IDisposable 的对象吗？**

**答：** 没错。IDisposable是专门为使用using语句量身定做的，增加一个using语句就像创建一个类的新实例，只不过总会调用其Dispose()方法。

**问：任何语句都能放在using块中吗？**

**答：** 当然可以。using的目的是帮助你确保用using语句创建的每一个对象都会撤销。不过，对这些对象做什么处理完全由你决定。实际上，如果用一个using语句创建一个对象，而在块中根本不使用这个对象，这也是完全可以的。不过，这没有任何意义，所以不建议你这样做。

**问：能不能在using语句之外调用Dispose()？**

**答：** 可以。并不总是需要使用using语句。如果已经用完一个对象，就可以自行调用Dispose()。或者可以完成所有必要的清理工作,比如手动地调用流的Close()方法。不过，如果使用了一个using语句，这会使你的代码更可读，更易于理解，而且可以避免由于对象未撤销而可能导致的问题。

**问：你提到了“try/finally”块，这是不是说，只有try和finally而没有catch也是可以的？**

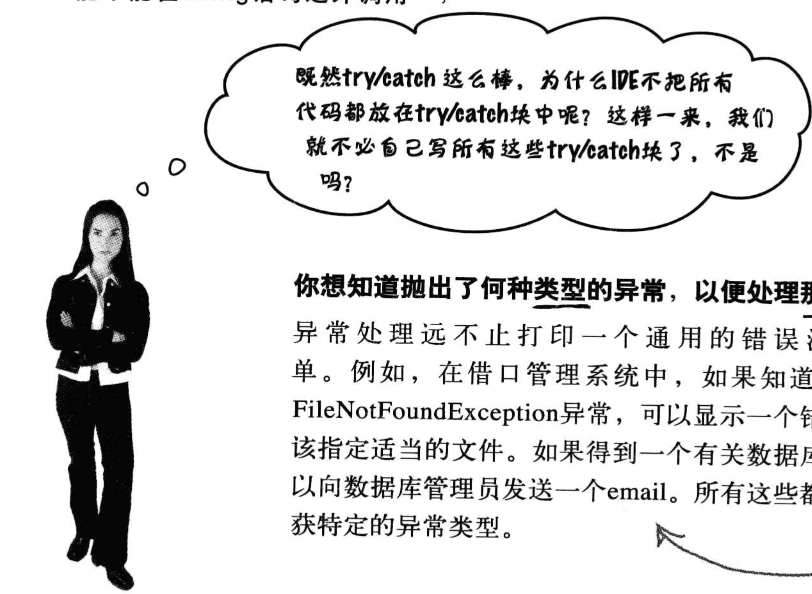
**答：** 确实如此!完全可以有一个try块而没有catch，然后是一个finally块。形式如下：

```
try {
      DoSomethingRisky();
      SomethingElseRisky();
}
finally {
      AlwaysExecuteThis();
}
```

如果SomethingRisky()抛出一个异常，finally块会立即运行。

**问：Dispose()只用于文件和流吗？**

**答：** 当然不是，有很多类都实现了IDisposable，如果使用了这样的类，就要使用一个using语句（后面几章会看到这样的一些类）。如果你要编写一个类，它必须以某种方式撤销，那么也可以实现IDisposable。




**你想知道抛出了何种类型的异常，以便处理那种异常。**

异常处理远不止打印一个通用的错误消息那么简单。例如，在借口管理系统中，如果知道得到了一个FileNotFoundException异常，可以显示一个错误，建议应该指定适当的文件。如果得到一个有关数据库的异常，可以向数据库管理员发送一个email。所有这些都取决于要捕获特定的异常类型。

正是因为这一点，才有那么多继承自Exception的异常类，你想编写自己的Exception子类也是出于这个原因。

# 史上最糟糕的catch块：全能型+注释

如果你愿意，catch块可以让你的程序继续运行。抛出一个异常后，捕获到这个异常，但是并不是结束程序，而是提供一个错误消息，接下来继续程序的运行。不过，有些时候，这不见得是一件好事。

来看以下的Calculator类，看起来它总有一些莫名其妙的行为，到底怎么了？

```
class Calculator {
...
  public void Divide(int dividend, int divisor) {
    try {
      this.quotient = dividend / divisor;
    } catch {
      // Note from Jim: we need to figure out a way to prevent
      // people from entering in zero in a division problem.
    }
  }
}
```

这里有问题。如果divisor是0，这会产生一个DivdeByZeroException。

不过这里有一个catch块。为什么还会有错误呢？

程序员认为通过使用一个空的catch块，就能把这个异常埋藏起来，但是这样做只能让以后查找问题的人更头疼。

## 应当处理异常，而不是埋藏起来

能够让程序继续运行，这并不表示已经处理了异常。在以上代码中，这个计算器不会崩溃…… 至少在Divide()方法中不会崩溃。但是，如果其他代码调用了这个方法，打印结果时会怎么样呢？如果divisor是0，这个方法可能会返回一个不正确的（而且意想不到的）值。

不要只是增加一个注释，然后把异常埋藏起来，需要处理这个异常。如果不能处理问题，也不要留下空catch块或者只有注释的catch块！这只会让跟踪代码的人更难办。最好是让程序继续抛出这些异常，因为这样一来就能很容易地看出哪里出了问题。

要记住，如果你的代码没有处理一个异常，这个异常会在调用栈中向上传递。让异常向上传递是一种完全合法的异常处理方法。

# 临时方案是允许的(权宜之计)

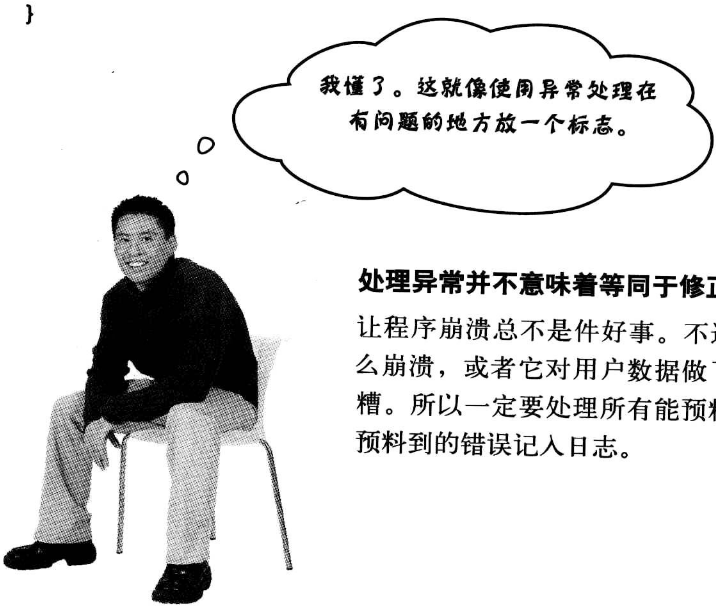
……但是在实际中，"临时"方案往往很糟糕地成为永久方案。

有时你发现了一个问题，而且知道这确实是个问题，但是不知道该怎么处理。在这些情况下，可能想把问题记下来，说明发生了什么。这虽然比不上真正地处理异常，但是总比什么也没有要好。

以下是对calculator的一个临时解决方案：

```
class Calculator {
...
  public void Divide(int dividend, int divisor) {
    try {
      this.quotient = dividend / divisor;
    } catch (Exception ex) {
      using (StreamWriter sw = new StreamWriter(@"C:\Logs\errors.txt");
         sw.WriteLine(ex.getMessage());
      };
    }
  }
}
```

这个代码还需要进一步修正，不过短期来看，这样可以指明哪里出了问题。另外，如果能得出为什么最初调用Divide方法时提供了一个为0的divisor不是更好吗？




**处理异常并不意味着等同于修正异常。**

让程序崩溃总不是件好事。不过，如果不知道它为什么崩溃，或者它对用户数据做了什么处理，这可能更糟。所以一定要处理所有能预料到的错误，并且把未预料到的错误记入日志。

## 关于异常处理的一些简单想法

设计代码时要妥善地处理失败。

要为用户提供有用的错误消息。

尽可能抛出.NET内置异常。只有当需要提供定制信息时才抛出定制异常。

要考虑到try块中的代码可能被短路。

……最重要的是……

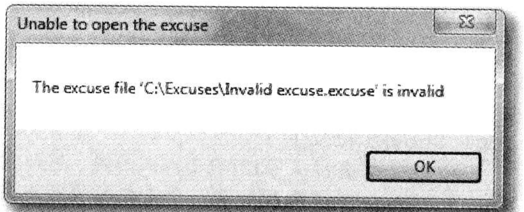
**要避免不必要的文件系统错误……只要使用流就应当使用USING块！切记 切记 切记！**

或者实现了IDisposable的对象。

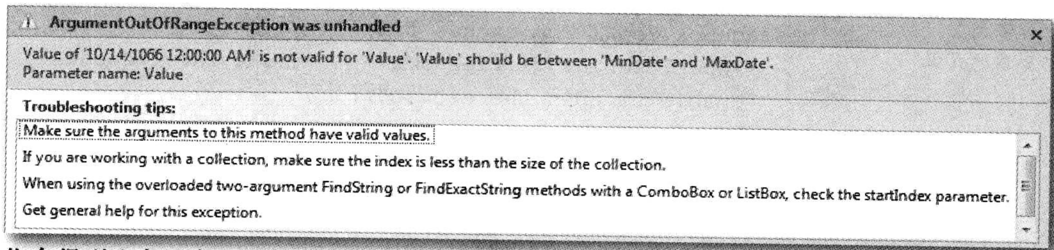



用前面学到的关于try/catch/finally的知识来改进Brian借口管理系统中的异常处理。

1. 为Open按钮的Click事件处理程序增加异常处理。只需建立一个简单的try/catch块弹出一个消息框。如果想打开一个文件，而它并不是一个真正的借口文件，应该弹出以下消息框：




2. 还不只如此，下面建立一个不合法的怪异的小借口文件。在Excuse.Save()方法的第一行放置一个断点，然后运行程序，保存一个借口。程序中断时，为**LastUsed**属性增加一个监视项。再在Watch窗口中编辑它的值，把它设置为**DateTime.Parse("October 14, 1066")**，你会看到属性值将更新为这个日期。让调试工具继续执行（按下F5）。你会得到这个ArgumentOutOfRange异常吗？




你会得到这个异常，这是因为窗体试图将DateTimePicker控件的Value属性设置为一个不于其MinDate的值。但是更重要的是，在它抛出异常之前，Excuse类已经写出了一个文件。这是一种非常有用的技术，一定要记住：可以使用已知不对的数据生成文件，这样以后可以用来测试你的程序。

3. 将这个包含错误数据的文件载入，应该会得到同样的异常。不过，如果试图打开一个文件，而它不是合法的借口文件，将会得到一个不同的异常。在第1步增加的异常处理中再嵌套一个异常处理块，确保试图加载一个非法的借口文件时（这种情况经常发生），程序不会失败。需要做到以下几点：

   1. 在try/catch块上面声明一个布尔变量，名为clearForm。如果有异常则将这个变量设置为true，以后检查这个变量来看窗体是否应当清空。
   2. 在Open按钮事件处理程序中增加的try/catch块内再嵌套增加一个try/catch块。
   3. 为外部try/catch块增加一个finally块，将窗体重置为其原来的空状态。如果clearForm变量设置为true，重置LastUsed.Value为DateTime.Now（这会返回当前日期）。




用前面学到的关于try/catch/finally的知识来改进Brian借口管理系统中的异常处理。

```
private void open_Click(object sender, EventArgs e) {
    if (CheckChanged()) {
        openFileDialog1.InitialDirectory = selectedFolder;
        openFileDialog1.Filter =
          "Excuse files (*.excuse)|*.excuse|All files (*.*)|*.*";
        openFileDialog1.FileName = description.Text + ".excuse";
        DialogResult result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK) {
            bool clearForm = false;
            try {
                currentExcuse = new Excuse(openFileDialog1.FileName);
                try {
                    UpdateForm(false);
                }
                catch (ArgumentOutOfRangeException) {
                    MessageBox.Show("The excuse file '"
                        + openFileDialog1.FileName + "' had a invalid data",
                        "Unable to open the excuse");
                    clearForm = true;
                }
            }
            catch (SerializationException ex) {
                MessageBox.Show("An error occurred while opening the excuse '"
                    + openFileDialog1.FileName + "'\n" + ex.Message,
                    "Unable to open the excuse", MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
                clearForm = true;
            }
            finally {
                if (clearForm) {
                    description.Text = "";
                    results.Text = "";
                    lastUsed.Value = DateTime.Now;
                }
            }
        }
    }
}
```

这是创建错误窗口的try/catch块，窗体调用Excuse构造函数加载一个借口时倘若出现问题，就会弹出一个错误窗口。

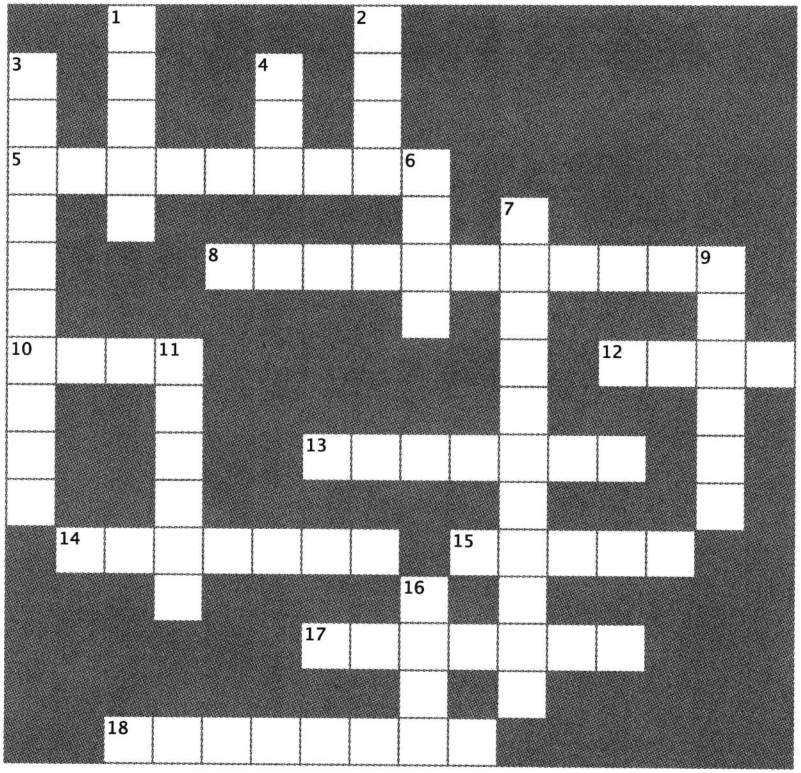
这是一个嵌套的try/catch。它处理另一个异常，可以加载文件但是文件中数据不合法时会导致这种异常。这不同于Excuse构造函数中出现的问题。

我们没有使用异常对象，所以catch语句在异常类型后面不需要一个变量名。

这是外层try/catch块弹出的消息框，这里显示了异常消息。

两个catch块都设置clearForm为true，所以这个finally块知道窗体应当重置。允许有代码与finally块交互，因为你知道finally块总会运行。

# 异常填字游戏



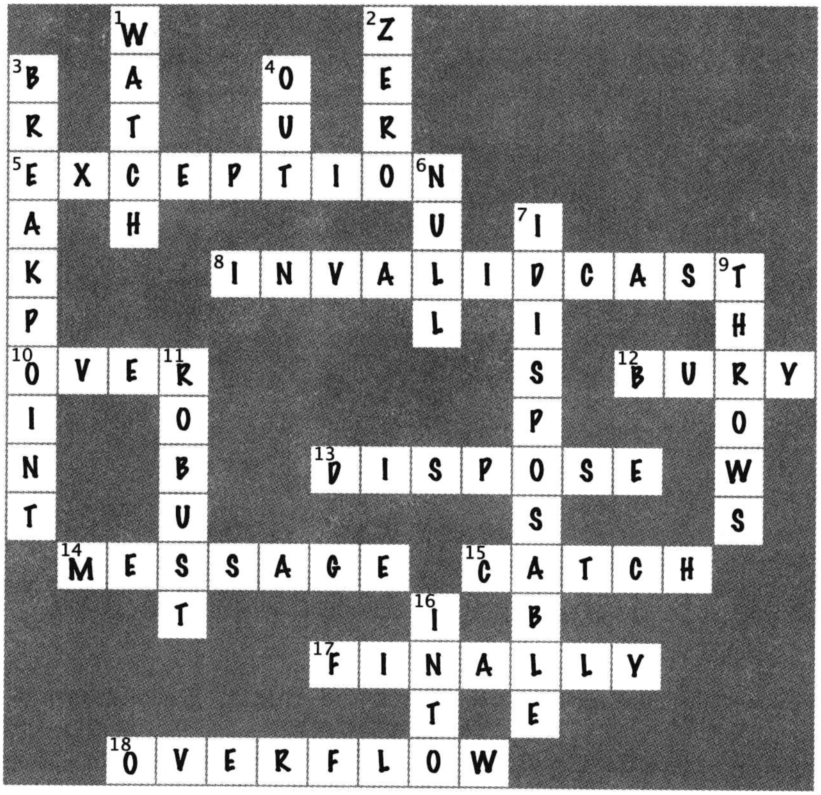

## 横向

5. DivideByZeroException和FormatException都继承的基类。
8. 要把一个值强制转换为一个无法容纳这个值的变量，此时会出现____________异常。
10. 如果下一条语句是一个方法，“Step______”会告诉调试工具执行这个方法中的所有语句，然后立即中断。
12. 如果__________你的异常，将很难跟踪。
13. 这个方法总在一个using块的最后调用。
14. Exception对象中的______字段包含一个描述串。
15. 一个try块可以有多个______块。
17. ______块包含处理完异常之后必须运行的语句。
18. 出现____________异常时说明你试图将一个过大的数硬塞到一个无法放下这个数的变量中。

## 纵向

1. IDE中的一个窗口，可以用来检查变量的值。
2. 如果要除以______，会得到一个异常。
3. 如果希望达到某行代码时调试工具停止执行，要设置____________。
4. “Step ______”告诉调试工具执行当前方法中的其余语句，然后中断。
6. 如果一个引用没有指向任何对象，则包含______。
7. 只有对象实现了____________接口时才能在一个using语句中为这个对象声明一个变量。
9. 如果一个语句存在一个问题，它会______一个异常。
11. 能妥善处理错误的程序。
16. 如果下一条语句是一个方法，“Step______”告诉调试工具执行该方法中的第一条语句。

# 异常填字游戏答案


1 WATCH
2 ZERO
3 BREAKPOINT
4 OUT
5 EXCEPTION
6 NULL
7 IDISPOSABLE
8 INVALIDCAST
9 THROWS
10 OVER
11 ROBUST
12 BURY
13 DISPOSE
14 MESSAGE
15 CATCH
16 INT
17 FINALLY
18 OVERFLOW

## Brian终于能度假了……

现在Brian在异常处理方面已经得心应手，他的工作干得很顺，终于得到了他应得的（而且老板认可的）假期。



## ……还有更棒的！

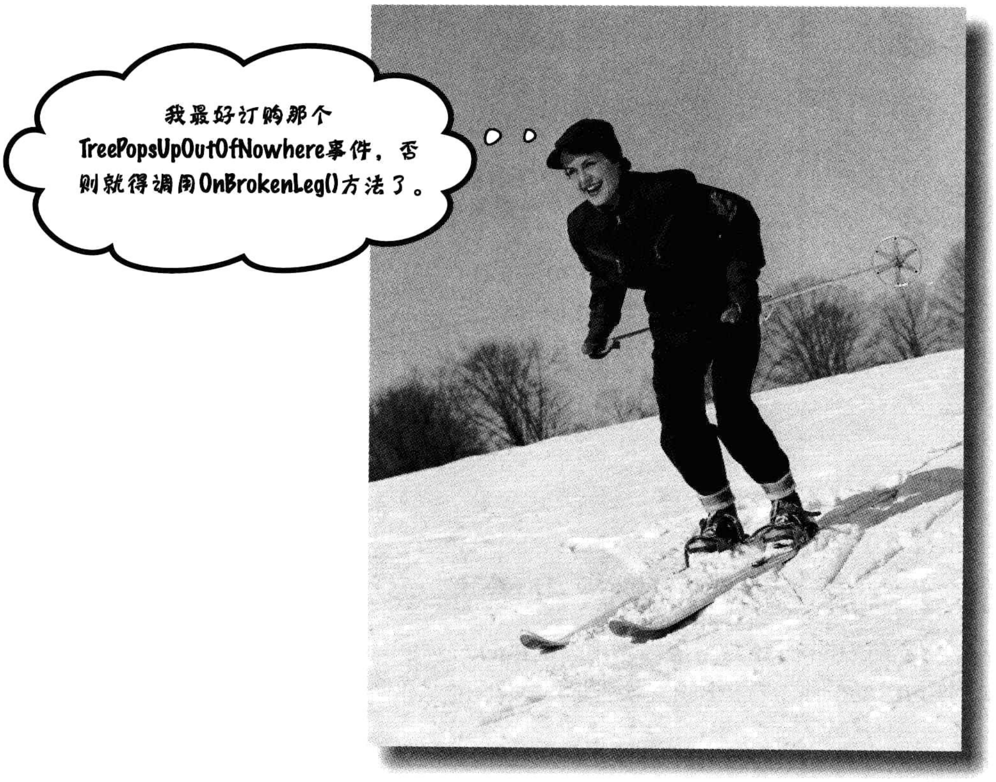
你的异常处理水平不只是能防止出现问题。还能确保Brian的老板根本不知道哪里会有问题：




**妥善的异常处理对用户来说是不可见的。程序绝对不会崩溃，如果出现问题，也能妥善地得到解决，而不是提供一些莫名其妙的错误消息。**

# 11 事件与委托

# 你不监视时代码在做什么




**对象开始自己考虑问题。**

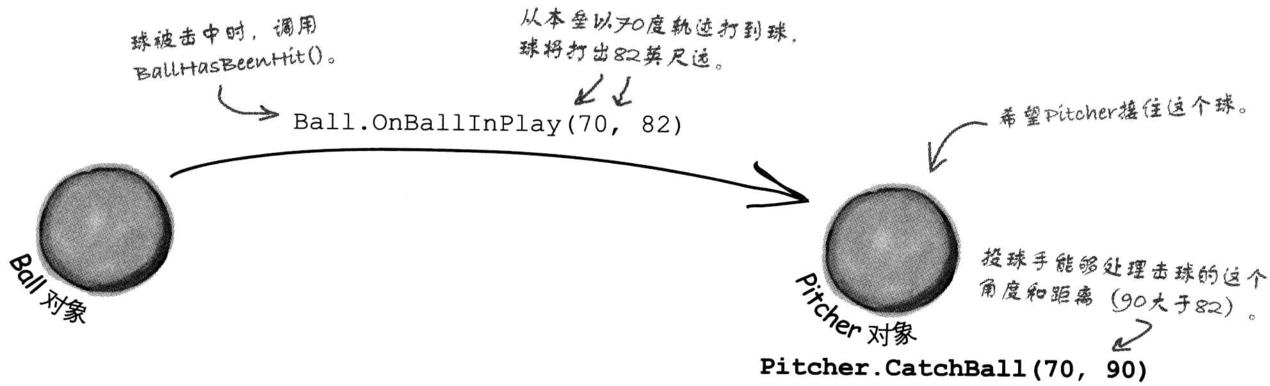
你并不总能控制对象在干什么。有时候……会发生一些事情。如果真是如此，你希望对象能足够聪明，对发生的事情做出响应。这正是事件要做的事情。一个对象发布一个事件，其他对象订购这个事件，大家联合起来保证工作顺利进行。听起来很不错，不过如果希望对象对于谁能监听有所控制，就不那么容易了。这里就需要用到回调。

# 希望你的对象自己考虑问题吗？

假设你在编写一个棒球模拟系统。你要为一场比赛建模，想把这个软件卖给Yankees棒球队，大挣一笔（他们确实财大气粗，不是吗）。你创建了Ball、Pitcher、Umpire和Fan对象，还有很多其他对象。另外还编写了代码使Pitcher对象能接住球。

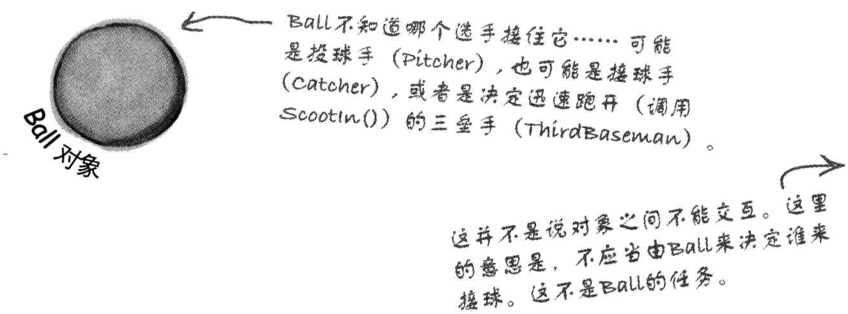
这是对方法命名的一种标准做法，后面还会谈到方法的命名。

现在只需要把所有对象连起来。为Ball增加一个OnBallInPlay()方法，接下来希望Pitcher对象用它的事件处理方法做出响应。一旦编写了这些方法，只需要把各个方法联合在一起：




# 不过一个对象怎么<u>知道</u>要做出响应呢？

这就是问题所在。你希望Ball对象只考虑被击中，而Pitcher对象只考虑接住击中的球。换句话说，你不希望Ball直接告诉Pitcher：^“我向你飞过来了”。




**你希望对象只考虑它自己，而不考虑其他对象。这里就<u>分离了</u>各个对象的<u>关注点</u>。**

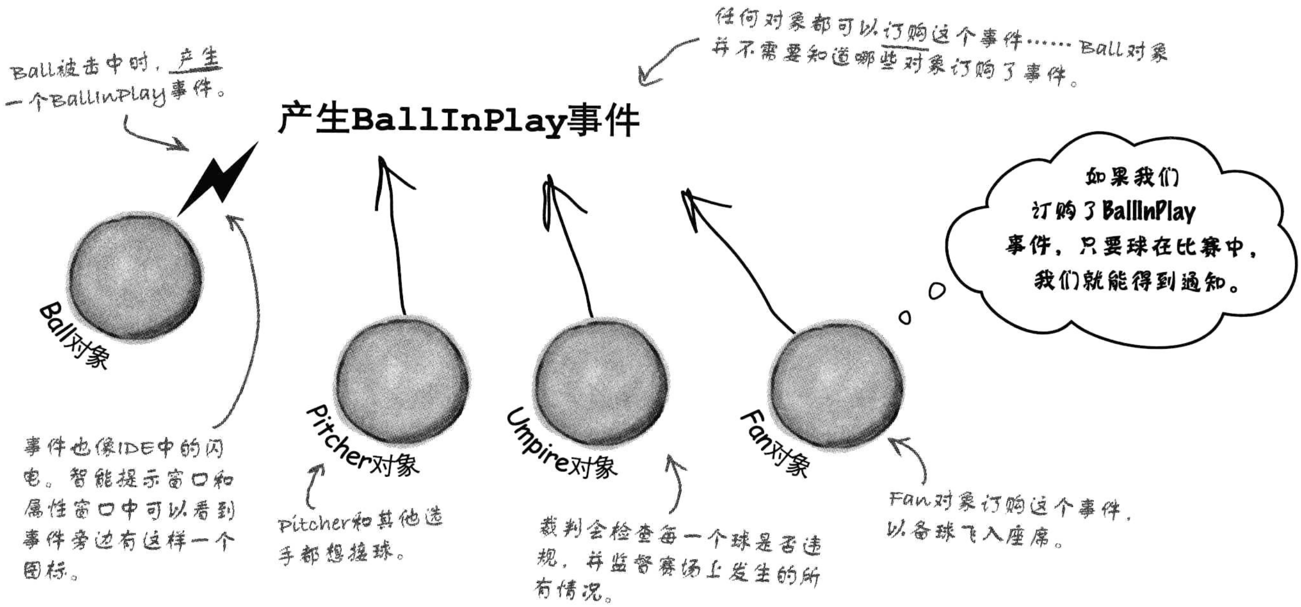
## 出现一个事件时…… 对象会监听

击中球时，就需要使用一个事件。事件（event）是程序中发生的某个事情。通过事件，其他对象就能对这个事件做出响应，比如Pitcher对象。

更棒的是，可以有不只一个对象监听事件。所以Pitcher可以监听击球事件，同样地，Catcher、ThirdBaseman、Umpire甚至Fan都可以监听这个事件。而且每个对象对这个事件的响应可以各不相同。

所以，我们只是希望Ball对象能产生一个事件。接下来，其他对象能订购这种特定类型的事件…… 也就是监听这个事件，事件发生时能得到通知。

**事件，名词。**
**发生的事情，特别是重要事件。日食是一个让人叹为观止的事件。**




## 想对事件做些什么吗?需要一个事件处理程序

一旦你的对象“听到”一个事件，可以创建一些针对这个事件运行的代码。这些代码称为事件处理程序（event handler）。事件处理程序得到事件的有关信息，每次事件发生时都会运行。

前面一直在这样做。每次点击一个按钮时，就会产生一个事件，你的代码会对这个事件做出响应。

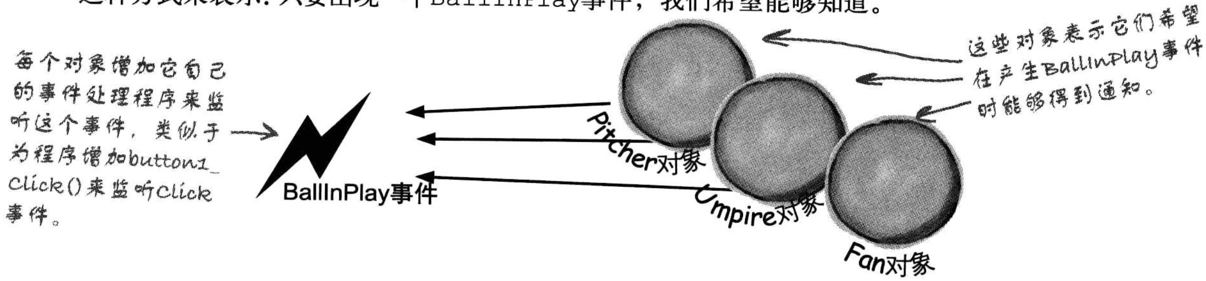
要记住，所有这些会在运行时发生，而无需你的干预。所以你可以编写代码来产生事件，然后编写代码处理这些事件，接下来启动应用运行。只要产生一个事件，事件处理程序就会激活……不需要你插手做任何工作。而且，最棒的是，对象有分离的关注点。它们只考虑自己，而不考虑其他对象。

# 一个对象产生事件，其他对象监听……

下面来看C#中的事件、事件处理程序以及订购是如何工作的。

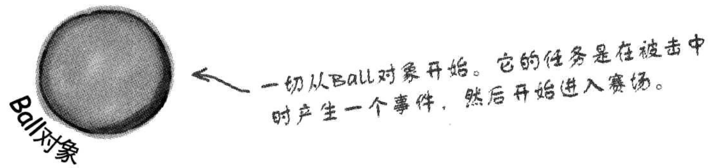
**① 首先，其他对象订购事件。**

Ball产生它的BallInPlay事件之前，其他对象需要订购这个事件。它们采用这种方式来表示：只要出现一个BallInPlay事件，我们希望能够知道。




**② 触发事件。**

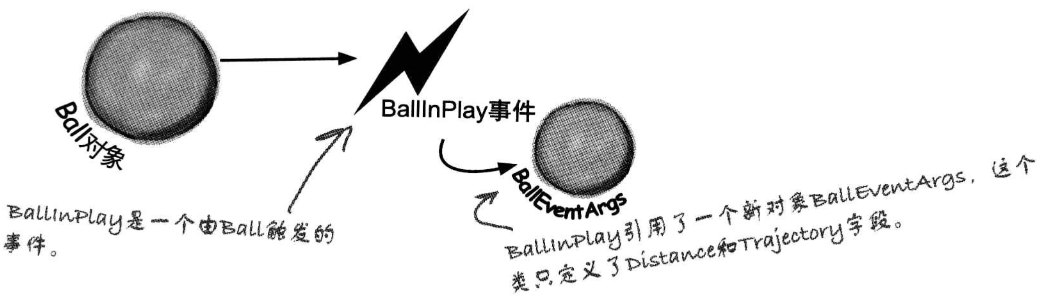
球被击中。此时Ball对象要产生一个新事件。




有时我们会说产生事件，或者触发事件，也可能说调用事件，这些都是一个意思。只不过说法不同。

**③ 球产生一个事件。**

创建了一个新事件（稍后会介绍如何创建事件）。这个事件还有一些参数，如球的距离和轨迹。这些参数作为EventArgs对象的一个实例关联到事件，事件发出后，所有监听对象都能得到这个事件。

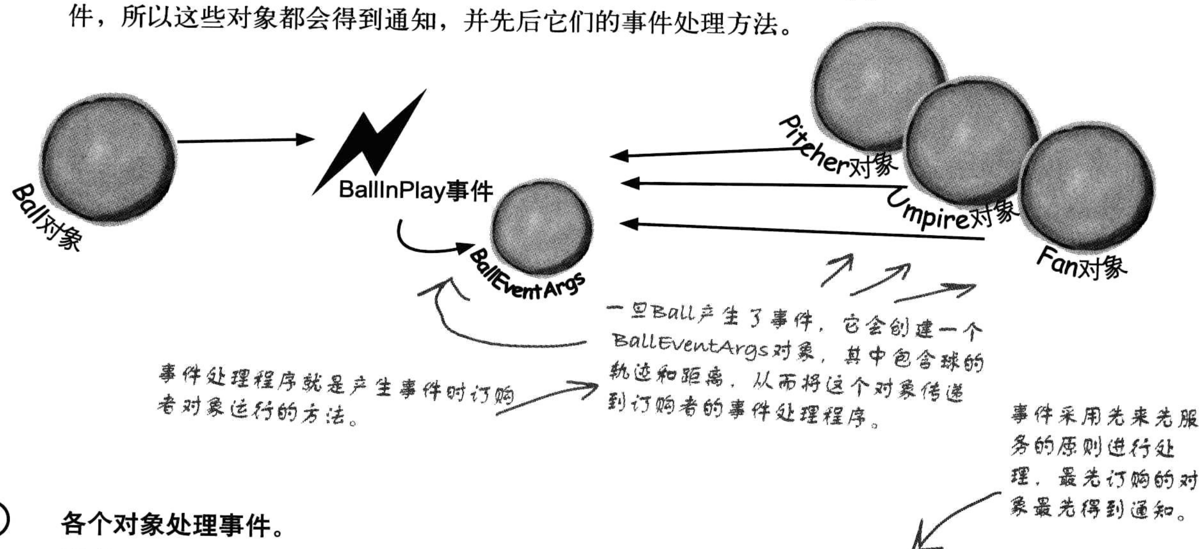


# 然后，其他对象处理这个事件

一旦事件产生，所有订购该事件的对象都会得到通知，并做一些处理。

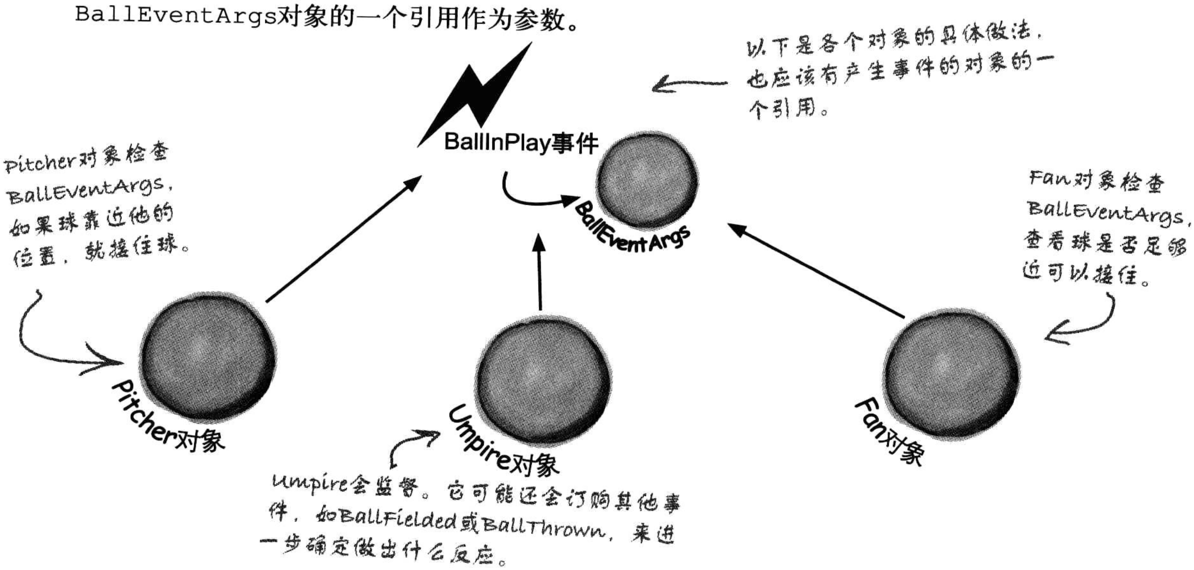
**④ 订购者得到通知。**

由于Pitcher、Umpire和Fan对象都订购了Ball对象的BallInPlay事件，所以这些对象都会得到通知，并先后它们的事件处理方法。




**⑤ 各个对象处理事件。**

现在，Pitcher、Umpire和Fan都会以它们自己的方式处理BallInPlay事件。不过并不是同时运行，它们的事件处理程序会先后得到调用，并以BallEventArgs对象的一个引用作为参数。

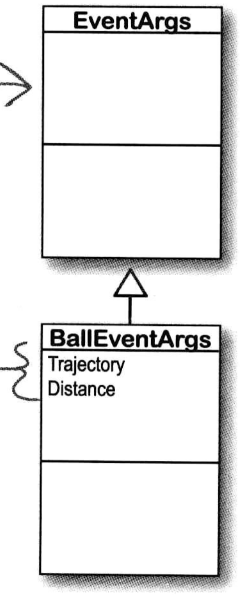


# 把各部分连接起来

既然已经对发生的情况有所了解，下面再来仔细地分析各部分如何集成。好在目前“部件”不多，不算复杂。

事件参数对象继承EventArgs是一个很好的想法（不过并不要求一定这样做）。EventArgs是一个空类，没有任何公共成员。

这表示，万一需要将EventArgs对象传递到一个不能处理这种特定类型的事件，可以对EventArgs对象完成向上强制转换。




球会使用这些属性向事件处理程序传递球在哪里被击中的有关信息。

**①** **需要一个表示事件参数的对象。**
要记住，`BallInPlay`事件有一些需要携带的参数。所以需要一个非常简单的对象来表示这些参数。为此.NET提供了一个标准类，名为**EventArgs**，不过这个类没有任何成员。它的目的只有一个，就是允许将你的事件参数对象传递到事件处理程序来使用。以下是事件参数类的声明:

```
class BallEventArgs : EventArgs
```

**②** **接下来需要在产生事件的类中定义这个事件。**
Ball类中有一行包含event关键字的代码，以此来通知其他对象存在这样一个事件，以便其他对象订购这个事件。这行代码可以放在类中的任何位置，通常比较靠近属性声明。不过，只要在Ball类中，不论这行声明放在什么地方，其他对象都能订购`Ball`的这个事件。事件的声明如下：

```
public event EventHandler BallInPlay;
```

事件总是公共的。这个事件在Ball类中定义，但我们希望Pitcher、Umpire等对象也能引用这个事件。如果希望只有这个类的其他实例能订购这个事件，可以把该事件置为私有。

event关键字后面是EventHandler。这不是C#的保留字，而是.NET提供的。之所以需要它，是为了告诉订购事件的对象它们的事件处理方法应该是什么样子。

使用EventHandler时，就是在告诉其他对象它们的事件处理程序应当有两个参数，一个是名为sender的object，另一个参数是名为e的EventArgs引用。sender是产生事件的对象的引用，e是EventArgs对象的一个引用。

**③ 订购类需要事件处理方法。**

订购Ball的BallInPlay事件的各个对象都需要有一个事件处理程序。其实你已经知道事件处理程序是如何工作的，每次增加一个方法来处理Button的Click事件或NumericUpDown的ValueChanged事件时，IDE都会为你的类增加一个事件处理方法。Ball的BallInPlay事件也不例外，它的事件处理程序看起来应该很熟悉：

```
void ball_BallInPlay(object sender, EventArgs e)
```

C#中并没有规则要求事件处理程序必须按某种方式命名，不过这是一种非常标准的命名约定：首先是对象引用名，后面是一个下划线，再后面是事件名。

BallInPlay事件声明中将其事件类型指定为EventHandler，这说明它需要两个参数，一个是名为sender的object，另一个是名为e的EventArgs，而且没有返回值。

包含这个特定事件处理方法的类中有一个Ball引用变量，名为ball，所以其BallInPlay事件处理程序以“ball_”开头，后面是处理的事件名“BallInPlay”。

**④ 各个对象订购事件。**

一旦建立了事件处理程序，Pitcher、Umpire、ThirdBaseman和Fan对象需要关联它们自己的事件处理程序。每个对象都有自己特定的ball _ BallInPlay方法，对这个事件做出不同的响应。所以，如果有一个Ball对象引用变量或字段，名为ball，可以用+=操作符来关联事件处理程序：

```
ball.BallInPlay += new EventHandler(ball_BallInPlay);
```

这告诉C#将这个事件处理程序关联到ball引用所指向的对象的BallInPlay事件。

+=操作符告诉C#为一个事件关联一个事件处理程序。

这部分指定为事件关联的事件处理方法。

这个事件处理方法的签名（参数和返回值）必须与EventHandler定义的签名一致，否则程序将不能编译。

未完，翻开下一页……

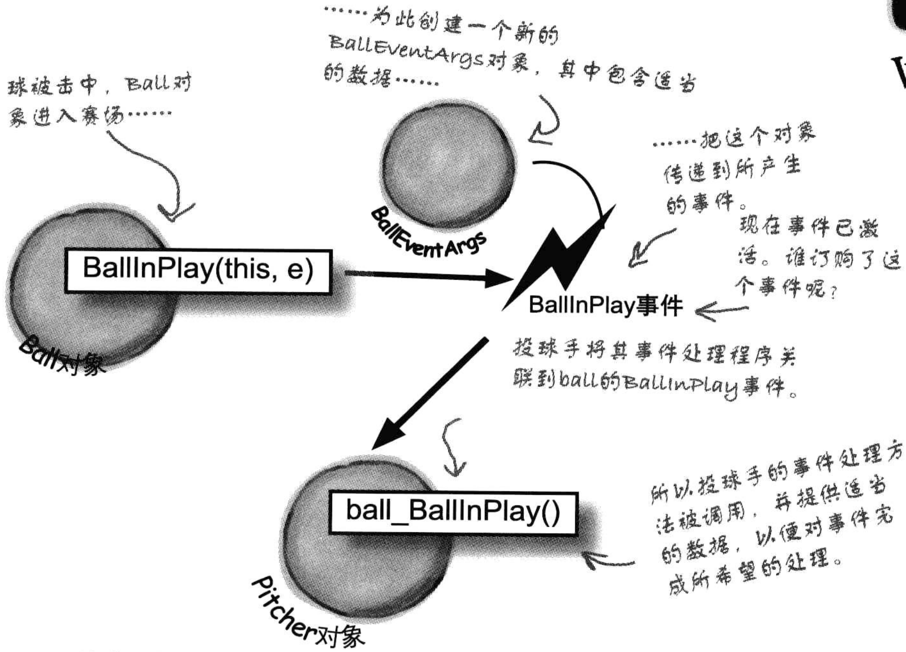
**⑤ Ball对象产生一个事件来通知订购者它已经进入赛场。**

既然已经创建事件，Ball可以产生这个事件作为模拟系统中其他事件的响应。产生一个事件很容易—只需调用BallInPlay事件。

```
EventHandler ballInPlay = BallInPlay;
if (ballInPlay != null)
    ballInPlay(this, e);
```

BallInPlay复制到一个变量ballInPlay，先检查它是否为null，再用它产生事件。

e是一个新的BallEventArgs对象。







**如果产生了一个事件，但是没有事件处理程序，会抛出一个异常。**

如果所有对象都没有为一个事件增加事件处理程序，事件处理程序则为null。所以一定要先检查，在产生事件之前确保事件处理程序不为null。如果没有做这个检查，就会抛出一个NullReferenceException。正是因为这个原因，在查看它是否为null之前，应当把事件复制到一个变量—在极少数情况下，有可能在检查事件处理程序是否为null和调用事件处理程序之间的某个时刻事件处理程序变成null。

## 增加产生事件的方法要使用一个标准名

用几分钟时间做个尝试，打开任意一个窗体的代码，在任意一个声明方法的位置上键入关键字override。键入空格后，就会弹出智能提示窗口：




注意到这些方法都有一个EventArgs参数了吗？产生事件时，它们都会将这个参数传递到事件。

Form对象可能产生很多事件，每个事件都有其自己的产生方法。窗体的OnDoubleClick()会产生DoubleClick事件，所以存在这样一个方法。因此Ball的事件也遵循同样的约定：它要有一个名为OnBallInPlay的方法，并取一个BallEventArgs对象作为参数。只要棒球模拟系统需要Ball产生BallInPlay事件,就会调用这个方法—所以当模拟系统检测到球棒击中球时，就会根据球的轨迹和距离创建BallEventArgs的一个新实例，并传递到OnBallInPlay()。

## there are no Dumb Questions

**问：声明事件时为什么需要加上EventHandler？我觉得事件处理程序应该是其他对象订购事件时使用的方法。**

**答：**确实如此。要订购一个事件，就要编写一个方法，这就是事件处理程序。不过，你注意到没有，我们在事件声明中（第2步）使用了EventHandler，另外在为事件处理程序订购事件的代码行中（第4步）也用到了EventHandler。EventHandler所做的就是定义事件的签名，它告诉订购这个事件的对象该如何定义事件处理方法。具体地，它指出，如果你想为一个方法订购这个事件，需要有两个参数（一个object和一个EventArgs引用），另外返回值类型为void。

**问：如果我想使用一个方法作为事件处理程序，但它与EventHandler定义的方法签名不一致会怎么样？**

**答：**你的程序将不能编译。编译器会确保你编写适当的事件处理方法，而不能用一个签名不匹配的方法来订购事件。也正是因为这个原因，标准事件处理程序EventHandler非常有用，只要看到它，你就能清楚地知道你的事件处理方法应该是什么样子。

**问：等一下，"标准"事件处理程序？还有其他的事件处理程序吗？**

**答：**没错！你的事件不一定发送一个object和一个EventArgs。实际上，事件可以发送任何内容，或者什么也不发送！请看上一页最下面智能提示窗口中的最后一行。注意到了吗？OnDragDrop方法需要一个DragEventArgs引用作为参数，而不是EventArgs引用。类似于BallEventArgs，DragEventArgs继承自EventArgs。窗体的DragDrop事件不使用EventHandler。它使用的是另一个事件处理程序DragEventHandler，如果想处理这个这个事件，事件处理方法就需要有一个object和一个DragEventArgs引用参数。

事件的参数由一个委托（delegate）来定义—EventArgs，和DragEventARGE就是委托的两个例子。不过稍后我们还会更详细地讨论委托。

**问：这么说来，我的事件处理程序也可以返回其他结果，而不是void，对吗？**

**答：**这样做是可以的，不过,这种想法通常并不好。如果事件处理程序不返回void，就不能把事件处理程序串起来。这意味着，不能向各个事件关联多个事件处理程序。由于事件处理程序串链是一个很方便的特性，所以最好还是从事件处理程序返回void。

**问：串起来？这是什么意思？**

**答：**这是指，多个对象可能订购同一个事件，就会把它们的事件处理程序一个接一个地串起来关联到这个事件。后面还会详细讨论这个内容。

**问：增加事件处理程序时使用+=就是因为这个原因吗？就好像把一个新事件处理程序增加到现有的事件处理程序后面？**

**答：**完全正确！增加事件处理程序往往会用+=。这样一来，你的事件处理程序就不会取代原有的事件处理程序。它只是成为一个可能很长的事件处理程序链中的一员，所有这些事件处理程序都监听同一个事件。

**问：为什么ball产生BallInPlay()事件时要使用"this"？**

**答：**因为这是标准事件处理程序的第一个参数。你注意到了吗？每个Click事件处理方法都有一个参数"object sender"。这个参数是产生这个事件的对象的一个引用。所以，如果处理一个按钮的点击事件，sender就指向被点击的按钮。如果处理一个BallInPlay事件，sender将指向进入赛场的Ball对象，所以ball产生事件时将这个参数设置为this。

**一个事件总由一个对象产生。**

**但是一个事件可以由多个对象响应。**

# IDE会为你自动创建事件处理程序

大多数程序员对事件处理程序命名时都遵循相同的命名约定。如果有一个Ball对象，它有一个BallInPlay事件，而且指向这个对象的引用名为ball，那么事件处理程序通常就命名为ball _ BallInPlay()。这不是一个必须严格遵守的规则，但是如果这样编写代码，其他程序员读起来就能容易得多。

幸运的是，利用IDE可以很容易地对事件处理程序适当地命名。IDE有一个特性，当你处理一个产生事件的类时，它能自动地为你增加事件处理方法。IDE能做到这一点应该并不奇怪。毕竟，在窗体中双击一个按钮时IDE就会自动创建事件处理程序。

**动手做**

1. **创建一个新的Windows应用，增加Ball和BallEventArgs。**
   Ball类如下：

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

以下是BallEventArgs类：

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance) {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}
```

2. **增加Pitcher的构造函数。**
   为工程增加一个新的Pitcher类。然后创建一个构造函数，取一个Ball引用作为参数，名为ball。这个构造函数中有一行代码将其事件处理程序增加到ball.BallInPlay。输入这行语句，不过先不要输入+=。

```
public Pitcher(Ball ball) {
    ball.BallInPlay
}
```

**3** **键入+=，IDE会为你完成这条语句。**

一旦在语句中键入+=，IDE就会显示一个非常有用的小框：

```
public Pitcher(Ball ball) {
    ball.BallInPlay +=
}                        new EventHandler(ball_BallInPlay);      (Press TAB to insert)
```

按下tab键后，IDE就会为你完成这条语句。语句如下：

```
public Pitcher(Ball ball) {
    ball.BallInPlay += new EventHandler(ball_BallInPlay);
}
```

在窗体设计工具中双击一个按钮时，IDE也会做同样的处理，自动地增加一个事件处理程序，只不过它还会向Form1.Designer.cs文件的InitializeComponent()方法增加代码，而不只是把它增加到类文件的最后。

**4** **IDE还会增加事件处理程序。**

到此还没有完—还需要增加一个方法关联到这个事件。幸运的是，这个工作IDE也会为你完成。

```
new EventHandler(ball_BallInPlay);
                 Press TAB to generate handler 'ball_BallInPlay' in this class
```

再次按下tab键，IDE会把这个事件处理方法增加到`Pitcher`类。IDE总是遵循`objectName_HandlerName()`命名约定：

```
void ball_BallInPlay(object sender, EventArgs e) {
    throw new NotImplementedException();
}
```

IDE总是填入这个NotImplementedException()作为一个占位代码，所以如果运行这个代码，它会抛出一个异常，告诉你还需要在它自动填入的地方做一些工作。

**5** **完成投球手的事件处理程序。**

既然已经有了可以增加到类的事件处理程序的骨架，填入其余的代码。投球手应当接住所有低球，否则就要防守一垒。

```
void ball_BallInPlay(object sender, EventArgs e) {
    if (e is BallEventArgs) {
        BallEventArgs ballEventArgs = e as BallEventArgs;
        if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
            CatchBall();
        else
            CoverFirstBase();
    }
}
```

由于BallEventArgs是EventArgs的一个子类，所以要使用as关键字完成向下强制转换，以便使用它的属性。

稍后就要增加这些方法。




现在要把你前面学到的知识付诸实践了。你的任务是完成Ball和Pitcher类，增加一个Fan类，并保证它们结合起来构成一个非常基本的棒球模拟系统。

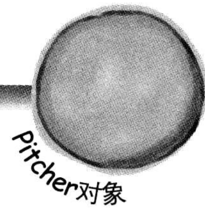
**1 完成Pitcher类。**

以下是我们已经提供的Pitcher代码。增加CatchBall()和CoverFirstBase()方法。这两个方法会打印出接球手已经接住球，或者迅速跑到一垒。

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
}
```

需要实现这两个方法向控制台写一行输出。




**2 编写一个Fan类。**

再创建一个类，名为Fan。Fan也应当在其构造函数中订购BallInPlay事件。球迷的事件处理程序应当查看球的距离是否大于400英尺，轨迹是否大于30（本垒打），如果确实满足这个条件就去去接球。如果不满足条件，球迷只是尖叫加油。将球迷的表现写到控制台。

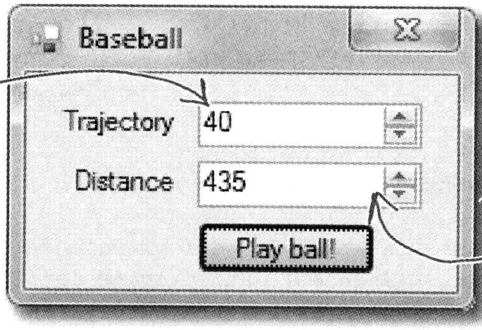
注意下一页上的输出窗口，看看它要输出些什么。



❸ **构建一个非常简单的模拟系统。**

创建一个新应用。这个应用应该有两个NumericUpDown控件：一个用来表示球的距离，另一个用来表示它的轨迹。增加一个按钮，标签为“Play ball!”。点击“Play ball!”按钮时，将按照两个NumericUpDown中的值击球。窗体应当如下所示：

trajectory的值范围为0到100，所以设置其Minimum属性为0，Maximum为100，Value设置为20。

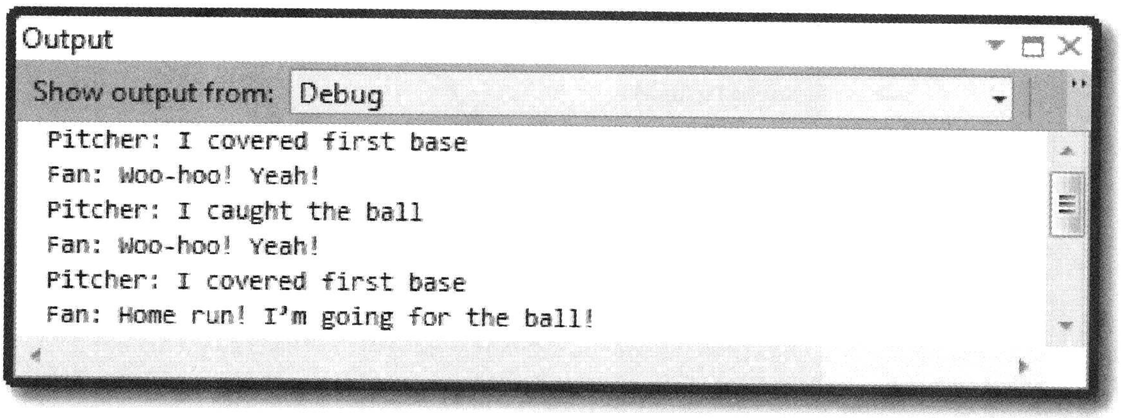



不要忘记使用Value属性前要把它强制转换为int值。

Distance范围为0到500，默认值为100。

❹ **生成以下输出。**

看看能不能让你的模拟系统生成以下输出，其中连续有3球入场。写出得到以下结果所用的值。




| Ball 1: | | Ball 2: | | Ball 3: | |
|---|---|---|---|---|---|
| 轨迹: | .................... | 轨迹: | .................... | 轨迹: | .................... |
| 距离: | .................... | 距离: | .................... | 距离: | .................... |

**Exercise Solution**

现在要把你前面学到的知识付诸实践了。你的任务是完成Ball和Pitcher类，增加一个Fan类，并保证它们结合起来构成一个非常基本的棒球模拟系统。

```
class Ball
{
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

OnBallInlay()方法只是产生BallInPlay事件，不过它必须先检查，确保BallInPlay不为null，否则会抛出一个异常。

```
class BallEventArgs : EventArgs
{
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance)
    {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}
```

只读的自动属性完全可以用作事件参数，因为事件处理程序只是读取传入的数据。

Fan对象的构造函数为BallInPlay事件增加事件处理程序。

```
class Fan {
    public Fan(Ball ball)
    {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e)
    {
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
                Console.WriteLine("Fan: Home run! I'm going for the ball!");
            else
                Console.WriteLine("Fan: Woo-hoo! Yeah!");
        }
    }
}
```

球迷的BallInPlay事件处理程序查找飞得又高又远的球。

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }

    private void CatchBall() {
        Console.WriteLine("Pitcher: I caught the ball");
    }

    private void CoverFirstBase() {
        Console.WriteLine("Pitcher: I covered first base");
    }
}

public partial class Form1 : Form {
    Ball ball = new Ball();
    Pitcher pitcher;
    Fan fan;

    public Form1() {
        InitializeComponent();
        pitcher = new Pitcher(ball);
        fan = new Fan(ball);
    }

    private void playBallButton_Click(object sender, EventArgs e) {
        BallEventArgs ballEventArgs = new BallEventArgs(
            (int)trajectory.Value, (int)distance.Value);
        ball.OnBallInPlay(ballEventArgs);
    }
}
```

这就是前面给出的Pitcher的BallInPlay事件处理程序。它查找飞得低的球。

窗体需要一个球、一个球迷和一个投球手。在构造函数中将球迷和投球手与球关联。

点击这个按钮时，窗体告诉投球手把球投向击球手，这就告诉球能发其BallInPlay事件，从而调用Pitcher和Fan对象的事件处理程序。

这是我们为得到输出所用的值。你使用的值可能与此稍有不同。

| | Ball 1: | Ball 2: | Ball 3: |
|---|---|---|---|
| 轨迹: | 75 | 48 | 40 |
| 距离: | 105 | 80 | 435 |

# 通用EventHandler允许你定义自己的事件类型

下面来看Ball类中的事件声明：

```
public event EventHandler BallInPlay;
```

现在来看按钮、窗体和你用过的大多数其他控件的Click事件声明：

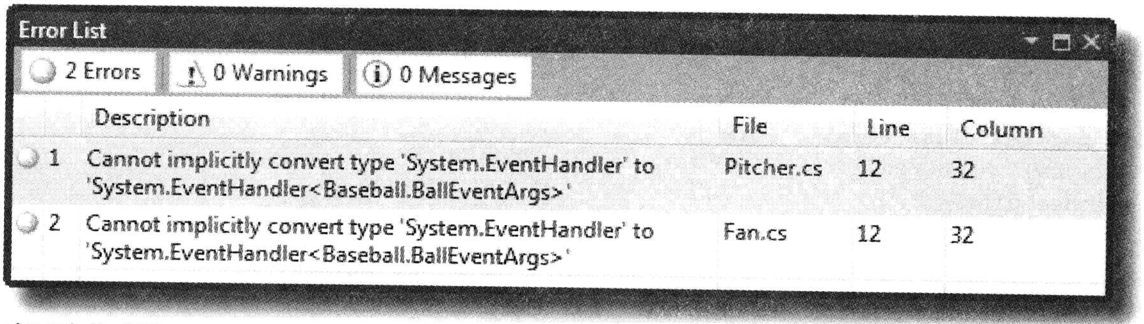
```
public event EventHandler Click;
```

注意到什么没有？他们的名字不同，但是都采用完全相同的方式声明。尽管这样是可以的，但查看类声明的人并不一定知道BallEventHandler在事件触发时总要传入一个BallEventArgs。幸运的是，.NET提供了一个很棒的工具可以很容易地传递这个信息：这就是EventHandler。修改ball的BallInPlay事件处理程序，如下所示：

EventHandler的通用参数必须是EventArgs的一个子类。

```
public event EventHandler<BallEventArgs> BallInPlay;
```

现在重新构建你的代码。应该能看到错误列表窗口中有两个错误：




既然改变了事件声明，Pitcher和Fan类也需要更新，通过向EventHandler传入这个通用参数从而关联到事件：

```
ball.BallInPlay += new EventHandler<BallEventArgs>(ball_BallInPlay);
```

## 省略new关键字和事件类型来使用隐式转换

如果像前面几页那样，使用IDE自动创建事件处理方法，它总会包含new关键字，后面跟有事件处理程序类型。不过，如果省略new关键字和事件处理程序类型，C#会完成一个隐式转换，为你得出类型：

```
ball.BallInPlay += ball_BallInPlay;
```

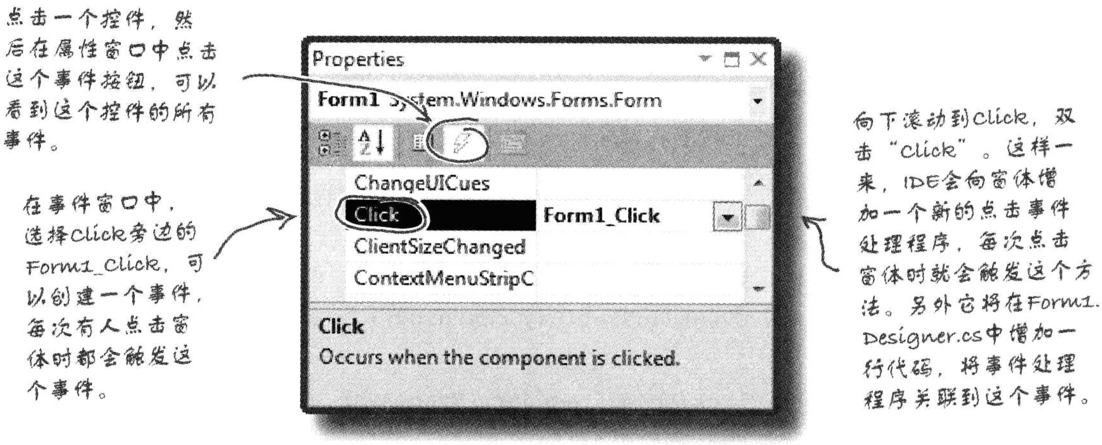
试着将Pitcher和Fan构造函数中的代码替换为上面这行代码。运行程序时，它也能很好地工作。

# 之前创建的窗体都使用了事件

如果创建一个按钮，然后在设计工具中双击这个按钮，并为类似button1 _ Click()的方法写代码，这就是在处理事件。

**动手做**

❶ 创建一个新的Windows Application工程。打开窗体，查看Properties窗口。还记得这个窗口最上方的图标吗？点击事件按钮（标有闪电的图标）。这会在Properties窗口中打开事件页：




❷ 双击事件页上的"Click"行。IDE会自动向窗体增加一个事件处理方法，名为Form1_Click。在这个方法中增加下面这行代码：

```
private void Form1_Click(object sender, EventArgs e) {
    MessageBox.Show("You just clicked on the form");
}
```

❸ 不过，Visual Studio不只是为你写一个简单的方法声明。它还会把事件处理程序与Form对象的Click事件关联起来。打开Form1.Designer.cs，使用IDE的快速查找特性（"Edit"→"Find and Replace"→"Quick Find"），在当前工程中搜索Form1 _ Click文本。可以找到下面这行代码：

```
this.Click += new System.EventHandler(this.Form1_Click);
```

现在运行程序，确保代码能正常工作！

未完，翻开下一页！ →

# 一个事件，多个处理程序

对于事件，有一点非常有用：可以把事件处理程序串起来，使得一个事件或委托可以一个接一个地调用多个方法。下面向应用中增加几个按钮，来看这是如何做到的。

**4** 向窗体增加两个方法：

```
private void SaySomething(object sender, EventArgs e) {
    MessageBox.Show("Something");
}
private void SaySomethingElse(object sender, EventArgs e) {
    MessageBox.Show("Something else");
}
```

**5** 现在再向窗体增加两个按钮。双击各个按钮增加它们的事件处理程序。以下是两个事件处理程序的代码：

```
private void button1_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomething);
}
private void button2_Click(object sender, EventArgs e) {
    this.Click += new EventHandler(SaySomethingElse);
}
```

继续下面的工作之前，先花点时间考虑一下这两个按钮会做什么。每个按钮为窗体的Click事件关联一个新的事件处理程序。在前面3步中，你像往常一样使用IDE来增加事件处理程序，使得每次窗体触发其Click事件时都弹出一个消息框，它为Form1.Designer.cs增加了代码，使用+=操作符关联其事件处理程序。

现在你增加了两个按钮，它们使用完全相同的语法为同一个Click事件串连了额外的事件处理程序。所以在继续之前，试着猜猜看如果运行这个程序，点击第一个按钮，然后点击第二个按钮，再点击窗体会发生什么。你能在运行程序之前得出结果吗？

**there are no Dumb Questions**

**问：** **向Pitcher对象增加一个新的事件处理程序时，为什么IDE会让它抛出一个异常？**

**答：** 它增加了代码来抛出一个NotImplementedException异常，这样做的目的是提醒你你还需要在这里实现代码。这是一个非常有用的异常，因为你可以像IDE一样把它用作为占位符。例如，需要构建一个类的骨架时，如果还不想填入所有代码，通常就会用到这个异常。这样一来，如果你的程序抛出这具异常，你就会知道这是因为你还需要完成代码，而不是因为你的程序出了问题。

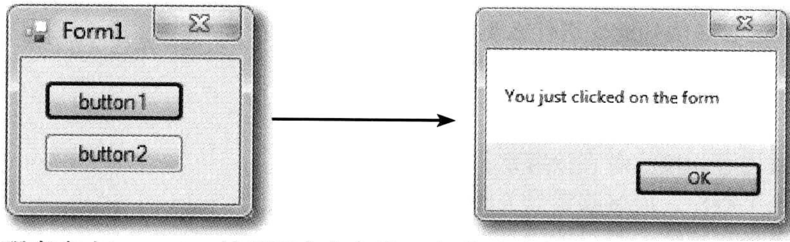



**事件处理程序需要“关联”。**

如果把一个按钮拖到窗体上，并增加一个名为button1_Click()的方法，尽管这个方法有正确的参数，但没有注册为监听这个按钮，那么这个方法不会得到调用。在设计工具中双击按钮时，IDE会发现默认的事件处理程序名已经被占用，所以它会为这个按钮增加一个名为button1_Click_1()的事件处理程序。

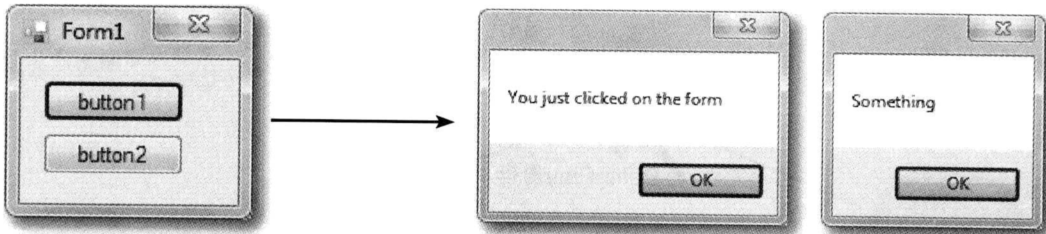
现在运行程序，完成以下步骤：

★ 点击窗体—会看到弹出一个消息框，指出“You just clicked on the form”。



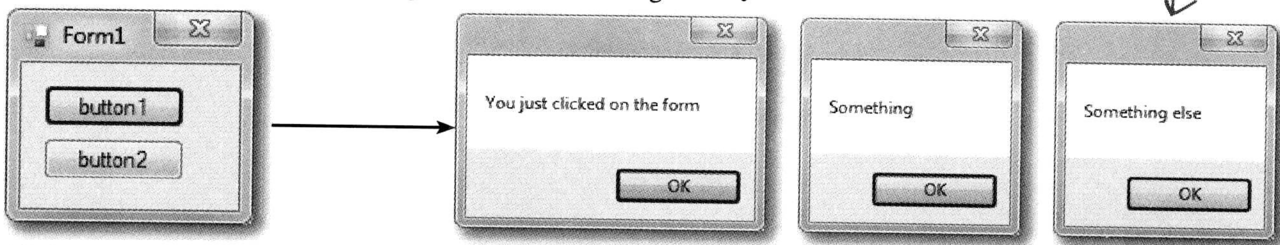

不出所料，窗体的Click事件处理程序弹出一个消息框。

★ 现在点击button1，然后再点击窗体。会看到弹出两个消息框：“You just clicked on the form”，然后是“Something”。




不过每次点击一个按钮时，它会导致下一次点击窗体时还会弹出另一个消息框！

★ 点击两次button2，然后再点击窗体。会看到4个消息框：“You just clicked on the form”，“Something”，“Something else”和“Something else”。

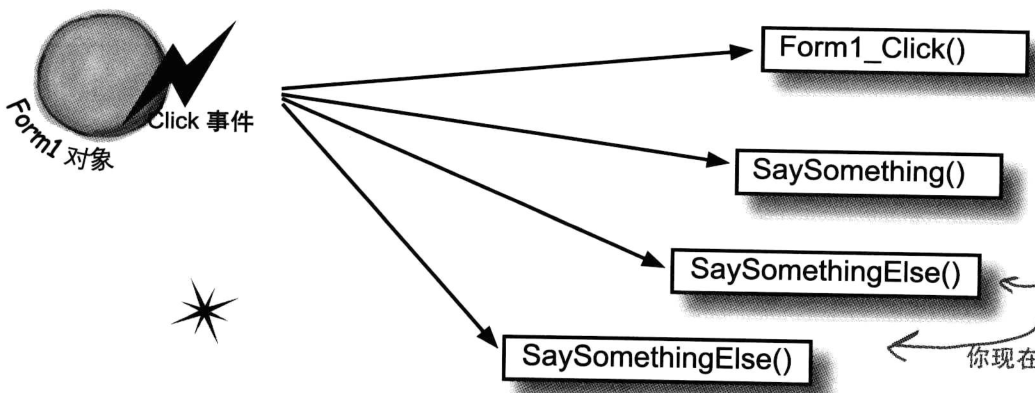



**发生了什么？**

每次点击其中一个按钮时，就把另一个方法（`Something()`或`SomethingElse()`）串连到窗体的Click事件上。如果一直点击按钮，就会一直为事件串连相同的方法。事件并不关心串连了多少个方法，甚至不关心这个链中是否有同一个方法出现了多次。只要事件触发，它就会按照增加方法时的顺序一个接一个地调用所有这些方法。

点击这些按钮时，它们会把不同的事件处理程序串连到窗体的Click事件。

这说明，点击按钮时你不会看到任何消息框弹出！需要点击窗体，因为按钮只是通过修改其Click事件来改变窗体的行为。

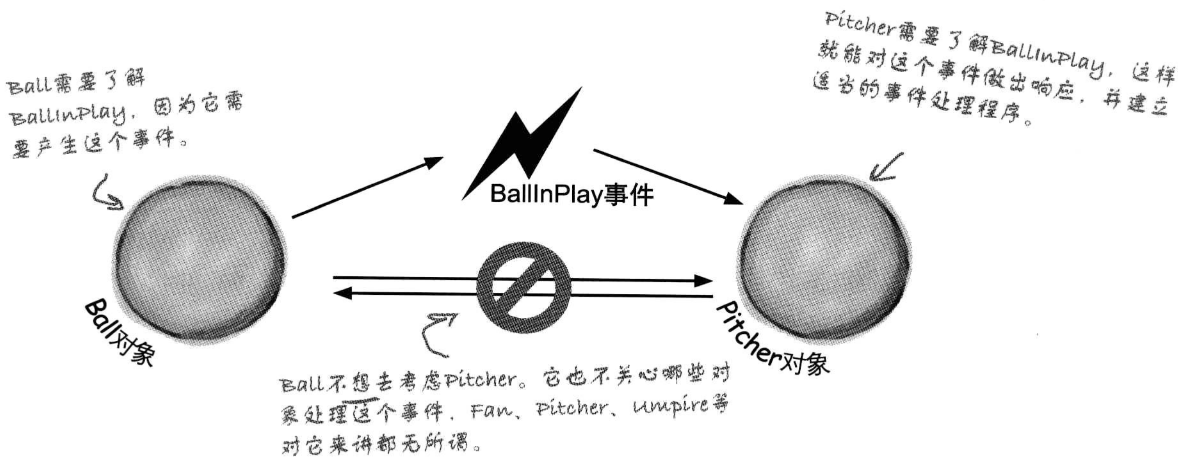



同一个方法可以多次链入一个事件。

# 连接事件发送者和事件接收者

关于事件，最难的问题之一是事件的发送者必须知道发送什么类型的事件，包括传入事件的参数。而事件的接收者必须了解其事件处理方法必须使用的返回类型和参数。

不过，注意这里是难点，不能直接把发送者和接收者直接绑定。你希望发送者发送事件，而不必操心是谁接收这个事件。而接收者只关心事件，而不在意是谁产生了这个事件。所以发送者和接收者都只关注事件，而不是对方。




## "我的人会与你的人联系"

你应该知道下面这个代码要做什么：

```
Ball currentBall;
```

它会创建一个引用变量，可以指向任何Ball对象。它并没有绑定到某一个Ball。相反，它可以指向任何ball对象，或者也可以是null，也就是根本不指向任何对象。

事件也需要一种类似的引用，只不过不是指向一个对象，它需要一个指向方法的引用。每个事件需要记录一组订购该事件的方法。你已经看到，这些方法可能在其他类中，甚至有可能是私有方法。所以事件怎么记录所有这些需要调用的事件处理方法呢？它使用了委托。

**del-e-gate，名词。**
派出或授权一个人代表其他人。总统向首脑会议派出一名代表。

## 委托代表一个具体的方法

事件最有用的一个方面是：事件触发时，它不知道在调用谁的事件处理方法。任何一个对象只要订购了事件，它的事件处理程序就会得到调用。那么事件对此该如何管理呢？

创建一个委托时，所要做的就是为它指向的方法指定方法签名。

所以这个委托可以用来引用任何取一个object和一个EventArgs参数而且没有返回值的方法。

事件使用了一个名为委托（delegate）的C#类型。委托是一种特殊的引用类型，可以指示一个类中的某个方法……委托是事件的基础。

其实这一章一直在使用委托！创建BallInPlay事件时，你使用了EventHandler。不错，EventHandler就是一个委托。如果在IDE中右键点击EventHandler，并选择“Go to definition”，你会看到下面的代码（你自己试试看）：

```
public delegate void EventHandler(object sender, EventArgs e);
```

这里指定了委托签名的返回值，表示EventHandler只能指向返回值类型为void的方法。

这个委托的名为EventHandler。

动手做

## 委托向工程增加一个新类型

为工程增加一个委托时，就增加了一个委托类型（delegate type）。用它创建一个字段或变量时，就是在创建该委托类型的一个实例。所以下面创建一个新工程。然后向这个工程增加一个新的类文件，名为ConvertsIntToString.cs。不过不在其中放入一个类，而只是增加一行代码：

```
    delegate string ConvertsIntToString(int i);
```

ConvertsIntToString是增加到工程的一个委托类型。现在可以用它来声明变量。这就像使用一个类或接口作为类型来定义变量一样。

接下来，为Program类增加一个名为HiThere()的方法：

```
private static string HiThere(int i)
{
    return "Hi there! #" + (i * 100);
}
```

这个方法的签名现在与ConvertsIntToString一致。

someMethod是类型为ConvertsIntToString的变量。它非常像一个引用变量，只不过不是在堆中的对象上放一个标签，而是在方法上放一个标签。

最后，填入Main()方法的代码：

```
static void Main(string[] args)
{
    ConvertsIntToString someMethod = new ConvertsIntToString(HiThere);
    string message = someMethod(5);
    Console.WriteLine(message);
    Console.ReadKey();
}
```

可以像设置其他变量一样设置someMethod。不过，把它作为一个方法来调用时，会调用它所指向的那个方法。

someMethod变量指向HiThere()方法。程序调用someMethod(5)时，它会调用HiThere()，并传入参数5，这会使它返回串值“Hi there! #500”，就像被直接调用一样。花点时间在调试工具中单步跟踪这个程序，来看到底做了什么。

# 委托的实际演练

委托并没有什么神奇的地方。实际上，委托使用的代码很少。下面使用委托帮助一个餐馆老板得到主厨的秘密配方。

动手做

**1** **创建一个新的Windows工程，并增加一个委托。**

委托总是出现在所有其他类外面，所以向工程增加一个新的类文件，命名为GetSecretIngredient.cs。其中只有一行代码：

```
delegate string GetSecretIngredient(int amount);
```

将类声明完全删除。这个委托可以用来创建一个变量，它能指向任何取一个int参数并返回string的方法。

**2** **为第一个主厨Suzanne创建一个类。**

Suzanne.cs中包含一个类，记录第一个主厨的秘密配方。它有一个私有方法SuzannesSecretIngredient()，这个方法的签名与GetSecretIngredient匹配。不过，这个类还有一个只读属性，请注意这个属性的类型。它返回一个GetSecretIngredient。所以其他对象可以使用这个属性来得到SuzannesIngredientList()方法的一个引用。

```
class Suzanne {
  public GetSecretIngredient MySecretIngredientMethod {
    get {
      return new GetSecretIngredient(SuzannesSecretIngredient);
    }
  }
  private string SuzannesSecretIngredient(int amount) {
    return amount.ToString() + " ounces of cloves";
  }
}
```

Suzanne的秘密配方方法取一个int参数amount，并返回一个string，这个串描述了她的秘密配方。

**3** **然后为第二位主厨Amy增加一个类。**

Amy的方法与Suzanne的方法类似：

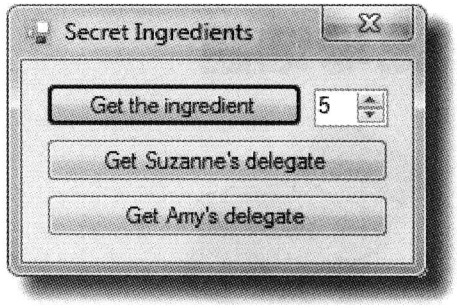
Amy的GetSecretIngredient属性返回GetSecretIngredient委托的一个新实例，指向Amy的秘密配方方法。

```
class Amy {
    public GetSecretIngredient AmysSecretIngredientMethod {
        get {
            return new GetSecretIngredient(AmysSecretIngredient);
        }
    }
    private string AmysSecretIngredient(int amount) {
        if (amount < 10)
            return amount.ToString()
                    + " cans of sardines -- you need more!";
        else
            return amount.ToString() + " cans of sardines";
    }
}
```

Amy的秘密配方方法也有一个int参数amount，并返回一个string，不过她返回的串不同于Suzanne返回的串。

**4** **创建一个新的Windows工程，并增加一个委托。**

创建这样一个窗体：




以下是窗体的代码：

```
GetSecretIngredient ingredientMethod = null;
Suzanne suzanne = new Suzanne();
Amy amy = new Amy();

private void useIngredient_Click(object sender, EventArgs e) {
    if (ingredientMethod != null)
        Console.WriteLine("I'll add " + ingredientMethod((int)amount.Value));
    else
        Console.WriteLine("I don't have a secret ingredient!");
}

private void getSuzanne_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(suzanne.MySecretIngredientMethod);
}

private void getAmy_Click(object sender, EventArgs e) {
    ingredientMethod = new GetSecretIngredient(amy.AmysSecretIngredientMethod);
}
```

**5** 使用调试工具查看委托是如何工作的。

你可以利用一个很棒的工具，这就是IDE的调试工具，要充分了解委托是如何工作的，这个工具确实很有帮助。完成以下步骤：

- 首先运行程序。先点击“Get the ingredient”按钮—这会向控制台写一行文本，指出“I don't have a secret ingredient”。
- 点击“Use Suzanne's delegate”按钮—这会将窗体的`ingredientMethod`字段（这是一个GetSecretIngredient委托）设置为Suzanne的GetSecretIngredient属性所返回的结果。这个属性返回GetSecretIngredient类型的一个新实例，指向`SuzannesSecretIngredient()`方法。
- 再点击“Get the ingredient”按钮。现在窗体的`ingredientMethod`字段指向`SuzannesSecretIngredient()`，它会调用这个方法，传入`numericUpDown`控件（确保这个控件命名为amount）中的值，并将输出写至控制台。
- 点击“Use Amy's delegate”按钮。它使用`Amy.GetSecretIngredient`属性来设置窗体的`ingredientMethod`字段，指向`AmysSecretIngredient()`方法。
- 再一次点击“Get the ingredient”按钮。现在它会调用Amy的方法。
- 现在使用调试工具来看到底发生了什么。在窗体中3个方法的第一行分别放置一个断点。然后重新运行程序（这会重置`ingredientMethod`，使它等于null），然后完成以上5个步骤。使用调试工具的Step Into (F11)特性单步跟踪每一行代码。观察点击“Get the ingredient”时发生了什么。它会跟踪进入Suzanne和Amy类，这取决于`ingredientMethod`字段指向哪个方法。

# 池塘谜题



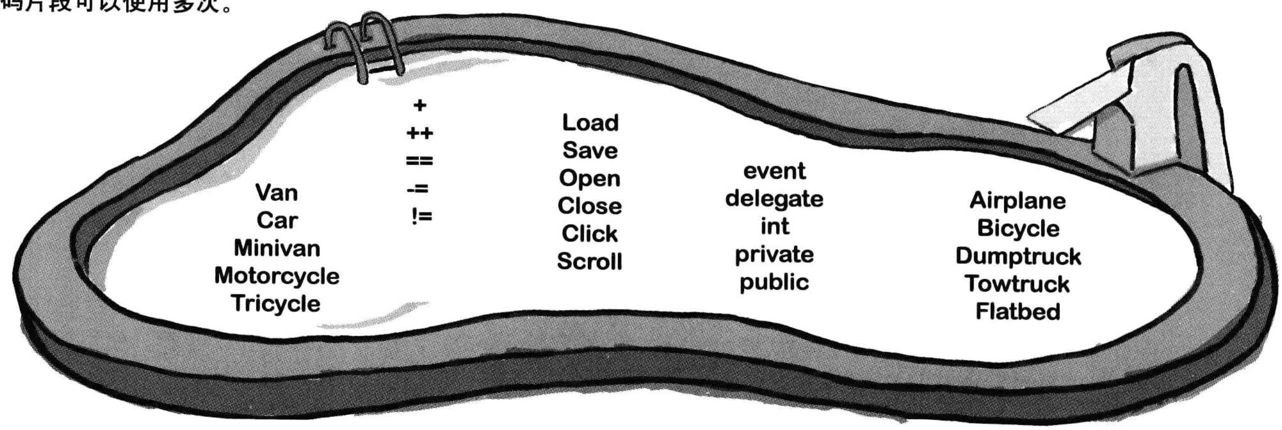
你的任务是从池塘里取出代码片段，在代码中填空。一个代码片段可以使用多次，另外并不是所有片段都要用到。你的目标是完成一个窗体的代码，点击窗体中的button1按钮时会向控制台写以下输出。

**输出**

**Fingers is coming to get you!**

```
public Form1() {
    InitializeComponent();
    this._______ += new EventHandler(Minivan);
    this._______ += new EventHandler(______________);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {
    button1._______ += new EventHandler(______________);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void ______________(object sender, EventArgs e) {
    button1._______ += new EventHandler(Dumptruck);
    button1._______ += new EventHandler(______________);
}
void ______________(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

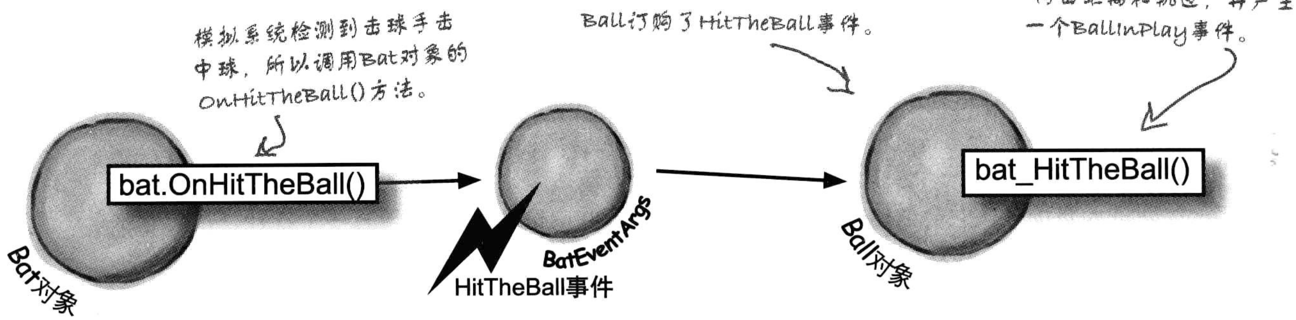
注意：池塘里的每个代码片段可以使用多次。



# 对象可以订购事件……

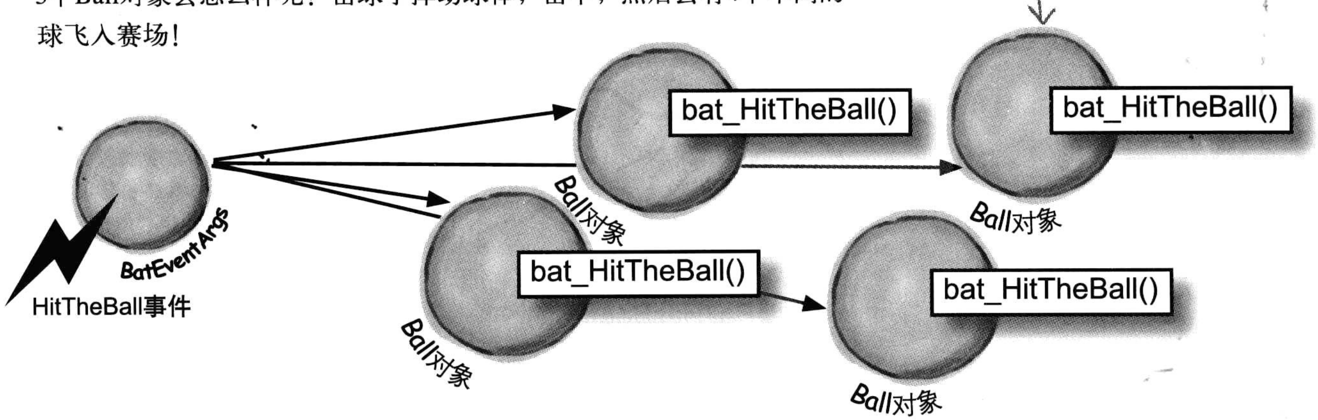
假设向模拟系统增加一个新类：Bat类，这个类又增加了一个HitTheBall事件。它的工作如下：如果模拟系统检测到选手击中球，会调用Bat对象的OnHitTheBall()方法，这会产生一个HitTheBall事件。

所以，现在可以向Ball类增加一个bat _ HitTheBall方法，订购Bat对象的HitTheBall事件。球被击中时，Ball的事件处理程序会调用自己的OnBallInPlay()方法，产生Ball自己的事件BallInPlay，这就会产生一系列连锁反应。接球手接球，球迷欢呼，裁判大叫…… 这才是真正的棒球比赛。




## ……但并不总是好事!

任何时刻球场里只能有一个球。但是如果Bat对象使用一个事件来宣布球被打中，那么任何Ball对象都能订购这个事件。这说明，我们遇到了一个棘手的问题，这里有一个小bug，如果一个程序员无意地又增加了3个Ball对象会怎么样呢？击球手挥动球棒，击中，然后会有4个不同的球飞入赛场！



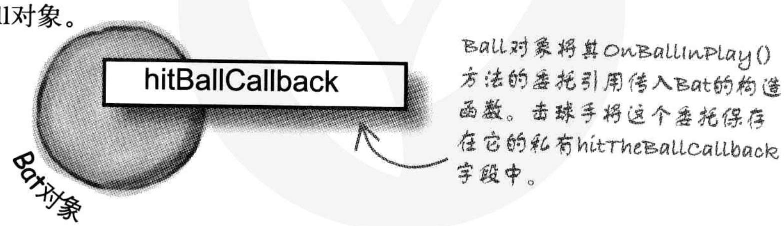
# 使用回调控制谁在监听

只有当仅有一个Ball和一个Bat时这个事件系统才能正常运作。如果有多个Ball对象，它们都订购这个公共事件HitTheBall，那么事件产生时所有这些球都会飞入赛场。不过，这肯定不合适…… 实际上只有一个Ball对象被击中。我们需要让投出的那个球关联到击球手，而且同时不允许其他球关联到击球手。

这里就可以使用回调（callback），这也是一种使用委托的方法。不是公布任何人都可以订购的一个事件，对象可以使用一个方法（通常是一个构造函数），取委托作为参数，把这个委托保存在一个私有字段中。我们将使用回调来确保Bat只会通知一个Ball。

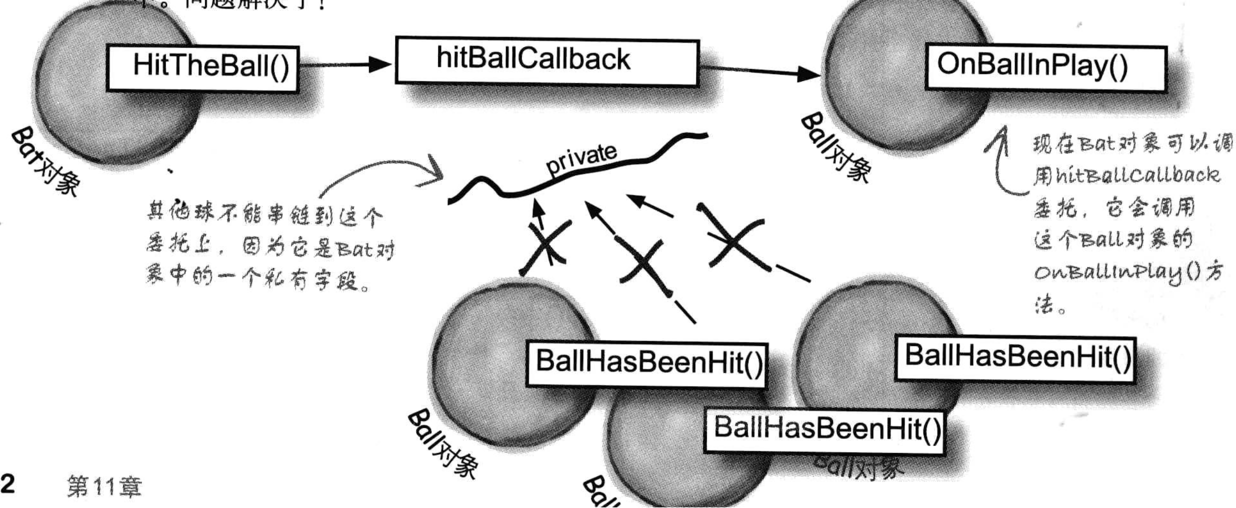
1. **Bat置其委托字段为私有。**
   要避免将不正确的Ball对象串链到Bat的委托上，最简单的办法就是让击球手将这个委托字段置为私有。这样一来，它就能对调用哪个Ball对象的方法有所控制。

2. **Bat的构造函数取一个委托，指向Ball中的一个方法。**
   球进入赛场时，它会创建击球手的一个新实例，并向这个Bat对象传入一个指针，指向它的OnBallInPlay()方法。这称为一个回调方法（callback method），因为Bat使用这个方法来回调实例化该Bat的Ball对象。




3. **击球手击中球时，会调用这个回调方法。**

   不过，由于击球手将这个委托置为私有，所以可以百分之百相信其他的所有球都不会击中。问题解决了！



## 黄金蟹事件

Henry “Flatfoot” Hodgkins是一个寻宝人（TreasureHunter）。他热衷于搜寻一种有翡翠外壳的半透明黄金蟹的踪迹，这是稀有水生动物珠宝市场中奖金最高的宝物之一。不过还有很多其他寻宝人。他们也在构造函数中得到了这个黄金蟹的引用，但Henry希望最先得到这笔奖金。

**5分钟揭秘**



在偷来的类图中，Henry发现每次有人靠近这个黄金蟹时，`GoldenCrab`类会产生一个`RunForCover`事件。更棒的是，这个事件包含`NewLocationArgs`，其中详细指出了黄金蟹会转移到哪里。不过，所有其他寻宝人都不知道这个事件，所以Henry认为他能捷足先登。

Henry在构造函数中增加了一些代码，把他的`ListenForClues()`方法注册为一个事件处理程序，来订购所得到的黄金蟹引用的`RunForCover`事件。然后，在黄金蟹后面布下一个眼线，掌握它逃走、藏起来以及产生`RunForCover`事件等动向，从而为Henry的`ListenForClues()`方法提供所需的全部信息。

一切都照计划进行，直到Henry找到黄金蟹的新位置。正当他急冲冲地去抓这只黄金蟹时，让他瞠目结舌的是，他看到另外3个寻宝人已经在那里了，正在为这个黄金蟹打作一团。

其他寻宝人是怎么战胜Henry先找到黄金蟹的？

答案见537页。

## 池塘谜题答案



构造函数将两个事件处理程序串到Load事件上。只要窗体加载就会触发这两个事件处理方法。

```
public Form1() {
    InitializeComponent();
    this.Load += new EventHandler(Minivan);
    this.Load += new EventHandler(Motorcycle);
}
void Towtruck(object sender, EventArgs e) {
    Console.Write("is coming ");
}
void Motorcycle(object sender, EventArgs e) {

    button1.Click += new EventHandler(Bicycle);
}
void Bicycle(object sender, EventArgs e) {
    Console.WriteLine("to get you!");
}
void Minivan(object sender, EventArgs e) {
    button1.Click += new EventHandler(Dumptruck);
    button1.Click += new EventHandler(Towtruck);
}
void Dumptruck(object sender, EventArgs e) {
    Console.Write("Fingers ");
}
```

两个Load事件处理程序将3个不同的事件处理程序串链到按钮的Click事件。

点击按钮时，它会调用串链到点击事件的3个事件处理程序。

# 回调只是一种使用委托的方式

回调是使用委托的另一种方式。这不是一个新的关键字或操作符。它只是描述了一种模式（pattern），通过这种方式对类使用委托，一个对象就可以告诉另一个对象："如果发生这件事请通知我，如果你方便的话！"

动手做

**① 在Baseball工程定义另一个委托。**

由于Bat有一个私有委托字段指向Ball对象的`OnBallInPlay()`方法，所以我们需要一个委托与这个方法的签名匹配：

```
delegate void BatCallback(BallEventArgs e);
```

委托并不总要放在单独的文件中。可以试着把这个定义放在Bat的同一文件中。

Bat对象的回调指向一个Ball对象的OnBallInPlay()方法，所以这个回调的委托需要与OnBallInPlay()方法的签名匹配，因此需要有一个BallEventArgs参数，而且返回值类型为void。

**② 向工程增加Bat类。**

`Bat`类很简单。它有一个`HitTheBall()`方法，每次击中一个球时模拟系统就会调用这个方法。这个`HitTheBall()`方法使用`hitBallCallback()`委托来调用球的`OnBallInPlay()`方法（或者传入Bat构造函数的任何方法）。

```
class Bat {
    private BatCallback hitBallCallback;
    public Bat(BatCallback callbackDelegate) {
        this.hitBallCallback = new BatCallback(callbackDelegate);
    }
    public void HitTheBall(BallEventArgs e) {
        if (hitBallCallback != null)
            hitBallCallback(e);
    }
}
```

一定要检查每个委托，确保它不为null，否则可能会抛出一个null引用异常。

**这里使用了=而不是+=，因为在这种情况下，我们只希望一个击球手监听某个球，所以这个委托只设置一次。不过，如果你确实想用+=也未尝不可，可以编写一个使用+=的回调来回调多个方法。回调的关键是完成调用的对象可以控制谁在监听。对于事件，其他对象可以通过增加事件处理程序来要求得到通知。对于回调，其他对象只需依靠委托，礼貌地请求得到通知。**

**③ 需要将击球手与球关联。**

那么`Bat`的构造函数如何得到某个球的`OnBallInPlay()`方法的引用呢？很容易，只需调用这个`Ball`对象的`GetNewBat()`方法，所以必须为Ball增加这样一个方法：

```
public Bat GetNewBat()
{
    return new Bat(new BatCallback(OnBallInPlay));
}
```

在Bat对象的构造函数中设置回调。不过在某些情况下，使用公共方法或者属性的设置存取方法来设置回调方法可能更合适。

Ball的GetNewBat()方法创建一个新的Bat对象，并使用BatCallBack委托将Ball自己的OnBallInPlay()方法引用传递到这个新击球手。这就是击球手击中球时要使用的回调方法。

**4 现在可以更好地封装Ball类。**

让一个产生事件的On...方法作为公共方法不太常见。可以自己试试看，打开窗体，试着调用playBall按钮的OnClick()事件。这是不允许的，因为这个方法是一个protected方法（所以子类可以覆盖这个方法）。下面对我们的Ball类也采用同样的模式，将Ball类的OnBallInPlay()方法置为protected：

```
protected void OnBallInPlay(BallEventArgs e) {
    EventHandler<BallEventArgs> ballInPlay = BallInPlay;
    if (ballInPlay != null)
        ballInPlay(this, e);
}
```

这是一个相当标准的模式，使用.NET类时会经常见到。如果一个.NET类有一个可以触发的事件，总能看到一个相应的以"On"开头的protected方法。

**5 接下来只需要关联窗体。**

窗体不能再调用Ball对象的OnBallInPlay()方法了—这正是我们所希望的。也正是因为这一点我们创建了Ball.GetNewBat()方法。现在窗体需要向Ball请求一个新的击球手，来击打这个球。此时，Ball对象要确保其OnBallInPlay()方法关联到击球手的回调。

```
private void playBallButton_Click(object sender, EventArgs e)
{
    Bat bat = ball.GetNewBat();
    BallEventArgs ballEventArgs = new BallEventArgs(
        (int)trajectory.Value, (int)distance.Value);
    bat.HitTheBall(ballEventArgs);
}
```

如果窗体（或模拟系统）希望击打一个Ball对象，需要从这个球得到一个新的Bat对象。这个球要确保回调关联到击球手。现在窗体调用击球手的HitTheBall()方法时，就会调用这个球的OnBallInPlay()方法，从而触发其BallInPlay事件。

现在运行程序，它的工作应该与以前一样，不过现在可以避免多个球监听同一个事件可能导致的问题。

不过不要只听我们讲，可以利用调试工具自己试试看！

## BULLET POINTS

- 向工程增加一个委托，就是在创建一个新类型，可以存储方法的引用。
- 事件使用委托来通知对象发生了某些动作。
- 如果对象需要对某个对象中发生的事情做出反应，可以订购这个对象的事件。
- EventHandler是一种委托，处理事件时这个委托很常用。
- 可以把多个事件处理程序串链到一个事件上。因此要使用+=为一个事件设置事件处理程序。
- 在使用事件或委托之前，一定要检查是否非null，以防止出现NullReferenceException异常。
- 工具条中的所有控件都使用事件来完成程序中的工作。
- 一个对象将一个方法的引用传入另一个对象，使它（只有它）能返回信息，这就称为一个回调。
- 利用事件，任何方法都可以匿名地订购对象的事件，而回调则允许对象对于所接收的委托有更多控制。
- 回调和事件都使用委托来引用和调用其他对象中的方法。
- 调试工具是一个很有用的工具，可以帮助你理解事件、委托和回调如何工作。要充分利用这个工具！

## there are no Dumb Questions

**问：回调与事件有什么区别？**

**答：** 事件和委托都是.NET的一部分。它们可以让一个对象向其他对象宣布发生了某个特定事情。对象发布一个事件时，可以有任意多个其他对象订购这个事件，而发布事件的对象不需要知道也不关心这些订购事件的对象。一个对象触发了一个事件时，如果有其他对象订购了这个事件，就会调用这些对象的各个事件处理程序。

回调根本不是.NET的一部分，实际上，"回调"只是一种使用委托的方法（或事件，完全可以使用一个私有事件来建立一个回调）。回调只是在两个类之间建立了一种关系，其中一个对象请求得到通知。与事件相比较，则是对象要求得到事件的通知。

**问：这么说来，回调并不是.NET中的具体类型？**

**答：** 对，回调不是类型。回调是一种模式，它只是以全新的方式使用了C#提供的现有类型、关键字和工具。可以再看看前面为击球手和球写的回调代码，能找到以前没用过的新关键字吗？找不到吧！不过它确实使用了一个委托，这是一个.NET类型。

有很多模式可以使用。实际上，编程中专门有一个领域叫做设计模式（design patterns）。你遇到的很多问题其实以前已经有人解决过，而且反复出现的问题都有相应的设计模式，可以供你使用。

**问：那么是不是说回调就是私有事件？**

**答：** 不完全是。看起来很容易这样想，但是私有事件是完全不同的东西。还记得private访问修饰符是什么含义吗？将一个类成员标志为private时，只有这个类的实例能访问它。所以，如果把一个事件标志为private，那么只有这个类的其他实例能订购这个事件。这与回调不同，因为作为私有事件，还是有可能有一个或多个对象匿名地订购这个事件。

**问：但是除了没有event关键字以外，它看起来确实像个事件，不是吗？**

**答：** 回调看上去之所以与事件如此相似，这是因为它们都使用了委托。而且它们都使用委托是有道理的，因为C#就是利用这个工具（委托）让一个对象向另一个对象传递某个方法的引用。

但是常规的事件和回调之间有一个重要区别，通过事件，一个类可以向全世界公布发生了某个特定的事情。而回调则相反，它不会发布。回调是私有的，而且能更好地控制谁能调用。

参考Head First Labs网站上的《Head First设计模式》一书。这是一个很好的途径，可以了解不同的模式，并应用到你自己的程序中。

www.headfirstlabs.com/books/hfdp/

你看到的第一个设计模式名为"发布者-订购者"模式，看起来应该不陌生吧。一个对象发布信息，其他对象订购这个信息。这看起来好像……




## 黄金蟹事件

### 其他寻宝人是怎么击败Henry先找到黄金蟹的？

奥秘就在于其他寻宝人如何搜寻他的信息来源。不过，先来看看Henry从偷来的类图中发现了什么。

在偷来的类图中，Henry发现每次有人靠近这个黄金蟹时，GoldenCrab类会产生一个RunForCover事件。更棒的是，这个事件包含NewLocationArgs，其中详细指出了黄金蟹会转移到哪里。不过，所有其他寻宝人都不知道这个事件，所以Henry认为他能捷足先登。

```
class GoldenCrab {
    public delegate void Escape(NewLocationArgs e);
    public event Escape RunForCover;
    public void SomeonesNearby() {
        NewLocationArgs e = new NewLocationArgs("Under the rock");
        RunForCover(e);
    }
}
class NewLocationArgs {
    public NewLocationArgs(HidingPlace newLocation) {
        this.newLocation = newLocation;
    }
    private HidingPlace newLocation;
    public HidingPlace NewLocation { get { return newLocation; } }
}
```

只要有人接近黄金蟹，它的SomeonesNearby()方法就会触发一个RunForCover事件，并找到一个藏身的地方。

那么Henry是怎么利用他新发现的内幕信息的呢？

Henry在构造函数中增加了一些代码，把他的ListenForClues()方法注册为一个事件处理程序，来订购所得到的黄金蟹引用的RunForCover事件。然后，在黄金蟹后面布下一个眼线，掌握它逃走、藏起来以及产生RunForCover事件等动向，从而为Henry的ListenForClues()方法提供所需的全部信息。

```
class TreasureHunter {
    public TreasureHunter(GoldenCrab treasure) {
        treasure.RunForCover += new GoldenCrab.Escape(treasure_RunForCover);
    }
    void treasure_RunForCover(NewLocationArgs e) {
        MoveHere(e.NewLocation);
    }
    void MoveHere(HidingPlace Location) {
        // ... code to move to a new location ...
    }
}
```

Henry以为他很聪明，修改了他的类构造函数来增加一个事件处理程序，使得每次黄金蟹产生RunForCover事件时就会调用他的MoveHere()方法。但是他忘了，其他寻宝人也继承同一个类，所以他自以为聪明的这些代码也会增加其他寻宝人的事件处理程序！

这就解释了为什么Henry的计划会适得其反。他向`TreasureHunter`构造函数增加事件处理程序时，无意地对所有寻宝人做了同样的处理！这说明，每个寻宝人的事件处理程序都会串链到同一个`RunForCover`事件。所以黄金蟹逃跑时，每个人都能得到这个事件的通知……如果Henry是第一个得到消息的倒还好，但是Henry不知道其他寻宝人何时得到通知，如果他们在Henry之前订购了这个事件，他们就会先得到事件通知。

## Sharpen your pencil

填空完成这个打地鼠游戏。需要提供回调代码。填完空后，在IDE中键入这些代码。或者可以先在IDE中编写完代码，再来填空。祝你玩得开心！

```
public partial class Form1 : Form {
    Mole mole;
    Random random = new Random();
    public Form1() {
        InitializeComponent();

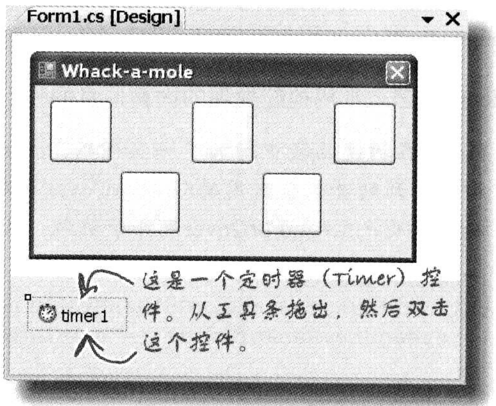
        mole = new Mole(random, new Mole.____________(____________________));
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void timer1_Tick(object sender, EventArgs e) {
        timer1.Stop();
        ToggleMole();
    }
    private void ToggleMole() {
        if (mole.Hidden == true)
            mole.Show();
        else
            mole.HideAgain();
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void MoleCallBack(int moleNumber, bool show) {
        if (moleNumber < 0) {
            timer1.Stop();
            return;
        }
        Button button;
        switch (moleNumber) {
            case 0: button = button1; break;
            case 1: button = button2; break;
            case 2: button = button3; break;
            case 3: button = button4; break;
            default: button = button5; break;
        }
        if (show == true)  {
            button.Text = "HIT ME!";
            button.BackColor = Color.Red;
        } else {
            button.Text = "";
            button.BackColor = SystemColors.Control;
        }
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
    private void button1_Click(object sender, EventArgs e) {
        mole.Smacked(0);
    }
}
```

窗体将指向一个回调方法的委托传入Mole的构造函数。填入这个委托。

双击窗体上的定时器控件时（先要把这个控件从工具条拖到窗体上），IDE会为它创建这个事件处理程序。定时器会反复触发Tick事件。下一章将介绍有关内容。

在定时器时间到时调用这个方法显示或隐藏地鼠。

这个switch确保适当的按钮改变颜色和文本。




这是一个定时器（Timer）控件。从工具条拖出，然后双击这个控件。

键入代码时，增加5个按钮事件处理程序。让button2_click()调用mole.Smacked(1)，然后让button3调用mole.Smacked(2)，button4调用mole.Smacked(3)，最后让button5调用mole.Smacked(4)。

只需在窗体设计工具中按常规双击这些按钮来增加事件处理程序。

```
using System.Windows.Forms;
class Mole {

    public ______________ void PopUp(int hole, bool show);

    private ____________ popUpCallback;
    private bool hidden;
    public bool Hidden { get { return hidden; } }
    private int timesHit = 0;
    private int timesShown = 0;
    private int hole = 0;
    Random random;

    public Mole(Random random, PopUp popUpCallback) {
        if (popUpCallback == null)
            throw new ArgumentException("popUpCallback can't be null");
        this.random = random;

        this.____________________ = ____________________;
        hidden = true;
    }

    public void Show() {
        timesShown++;
        hidden = false;
        hole = random.Next(5);

        ____________________(hole, true);
    }

    public void HideAgain() {
        hidden = true;

        ____________________(hole, false);
        CheckForGameOver();
    }

    public void Smacked(int holeSmacked) {
        if (holeSmacked == hole) {
            timesHit++;
            hidden = true;
            CheckForGameOver();

            ____________________(hole, false);
        }
    }

    private void CheckForGameOver() {
        if (timesShown >= 10) {
            popUpCallback(-1, false);
            MessageBox.Show("You scored " + timesHit, "Game over");
            Application.Exit();
        }
    }
}
```

填入委托和保存委托的字段，它们放在Mole类的最前面。

在这里确保回调不为null，如果为null，Mole对象抛出一个ArgumentException。

窗体创建一个新的Mole对象时，会传入其回调方法的一个引用。请仔细查看窗体，分析构造函数是如何调用的，然后填空。

地鼠显示之后，需要调用窗体显示地鼠的方法，将按钮变为红色，并显示文本"HIT ME!"。

HideAgain()和Smacked()方法也使用回调委托来调用窗体的这个方法。

这个游戏的做法是，使用定时器等待随机的一段时间（0.5～1.5秒之间）。一旦时间到，就告诉地鼠出现。窗体为Mole对象提供一个回调，Mole对象用这个回调来告诉窗体在某个地洞中（共5个地洞）显示或隐藏地鼠。然后，窗体再利用其定时器等待随机的一段时间（0.5～1.5秒之间），再告诉地鼠藏起来。

地鼠现身10次后游戏结束。你的得分就是打中地鼠的次数。

## Sharpen your pencil Solution

填空完成这个打地鼠游戏。需要提供回调代码。填完空后，在IDE中键入这些代码。或者可以先在IDE中编写完代码，再来填空。祝你玩得开心！

```
public partial class Form1 : Form {
    private void Form1_Load(object sender, EventArgs e) {

        mole = new Mole(random, new Mole.PopUp(MoleCallBack));
        timer1.Interval = random.Next(500, 1000);
        timer1.Start();
    }
}

class Mole {

    public delegate void PopUp(int hole, bool show);

    private PopUp popUpCallback;
 ...

    public Mole(Random random, PopUp popUpCallback) {
        this.random = random;
        this.popUpCallback = popUpCallback;
        hidden = true;
    }

    public void Show() {
        timesShown++;
        hidden = false;
        hole = random.Next(5);
        popUpCallback(hole, true);
    }

    public void HideAgain() {
        hidden = true;
        popUpCallback(hole, false);
        CheckForGameOver();
    }

    public void Smacked(int holeSmacked) {
        if (holeSmacked == hole) {
            timesHit++;
            hidden = true;
            CheckForGameOver();
            popUpCallback(hole, false);
        }
```

在这里窗体将其MoleCallBack()方法的引用传入Mole对象，从而允许地鼠调用窗体的方法。

地鼠在这里定义它的委托，并使用这个委托来建立一个私有字段，保存窗体上改变按钮颜色的方法的引用。

窗体创建Mole对象的一个新实例时，它将其MoleCallBack()方法的一个引用作为参数传入Mole的构造函数。构造函数中的这行代码将这个引用复制到其popUpCallback字段。这样一来，Mole的方法就可以使用这个字段来调用窗体的MoleCallBack()方法了。

地鼠现身、隐藏或者被打中时，Mole对象使用其popUpCallback委托字段调用窗体的一个方法来改变某个按钮的颜色和文本。

# 12 复习与预习

# 知识，能力，动手实践



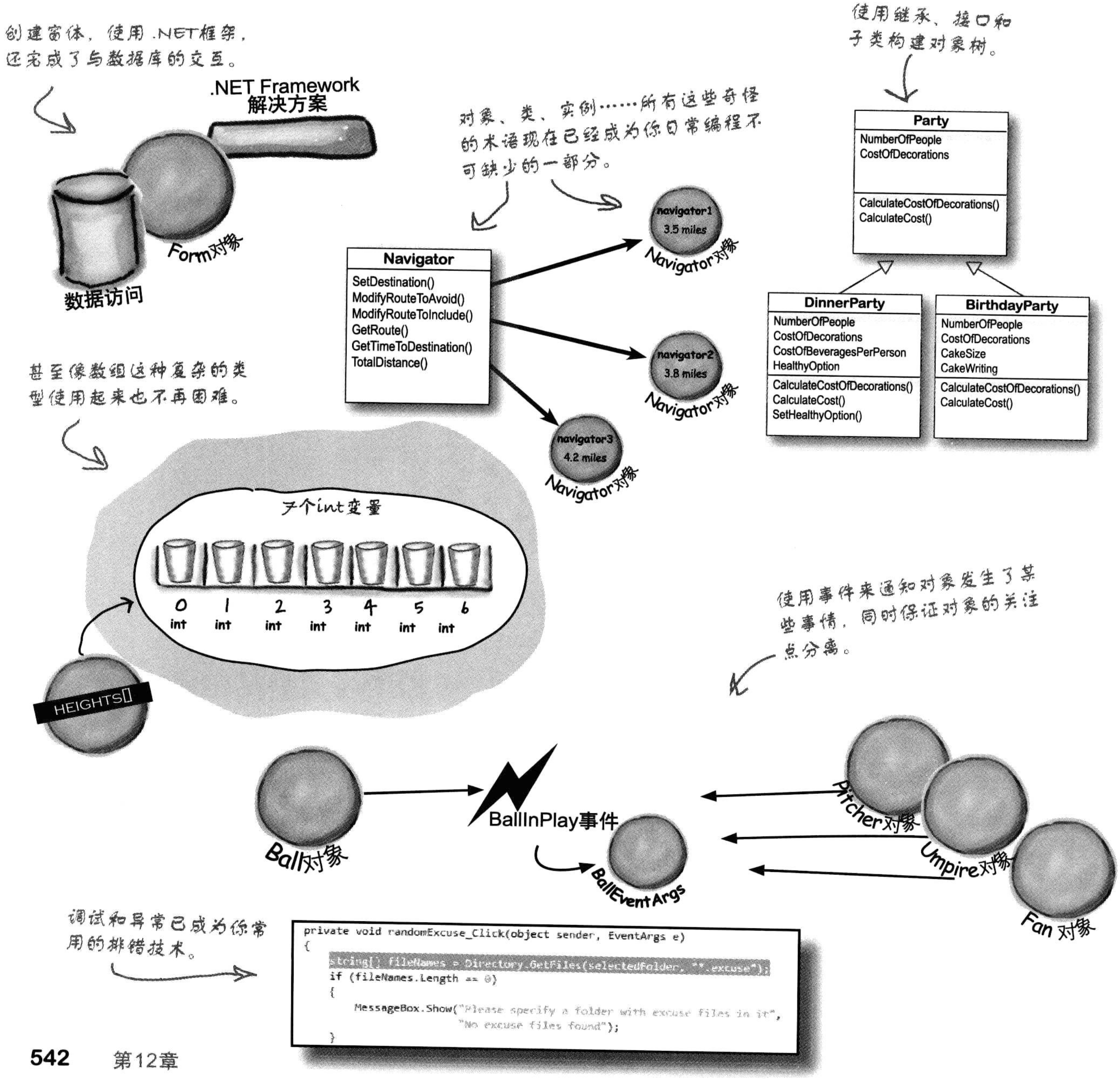

**光学不练是没有意义的，还需要真正动手去实践。**

除非你自己动手写一些真正能运行的代码，否则很难相信你能真正掌握C#中那些深奥的概念。这一章中，我们将使用前面学到的知识动手实践，还会预习后面很快要讲到的一些新知识。我们会构建一个相当复杂的应用，确保你真正掌握了前面各章学到的内容。所以，做好准备……现在就来构建真正的软件。

# 你已经学了不少了，~~伙计~~

[来自人力资源部的提醒："伙计"(baby)在正式场合讲很不准确。请使用其他说法，以免冒犯读者]

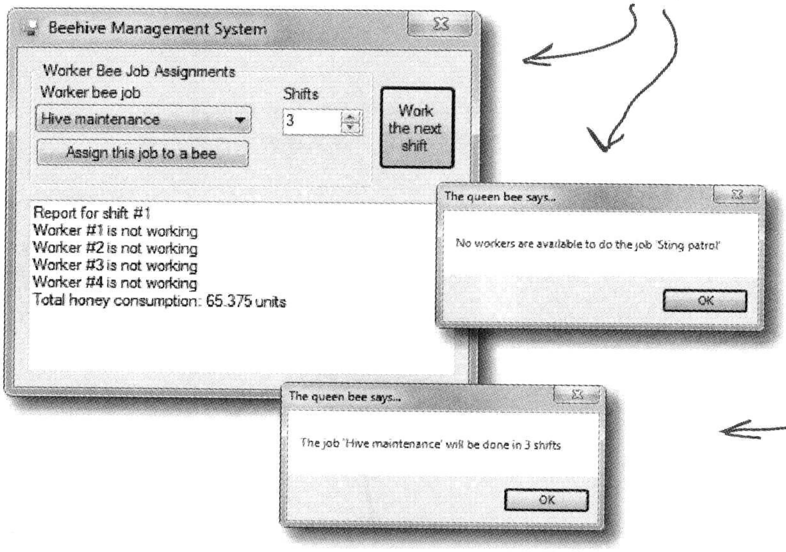
从最早使用IDE来帮助挽救Objectville纸业公司以来，我们已经学到了不少。在前面的几百页里，我们做了这样一些事情：



# 我们还是不错的养蜂人

翻回到第6章，我们构建了一些蜜蜂类，还记得吗？

不同的蜜蜂做不同的工作……



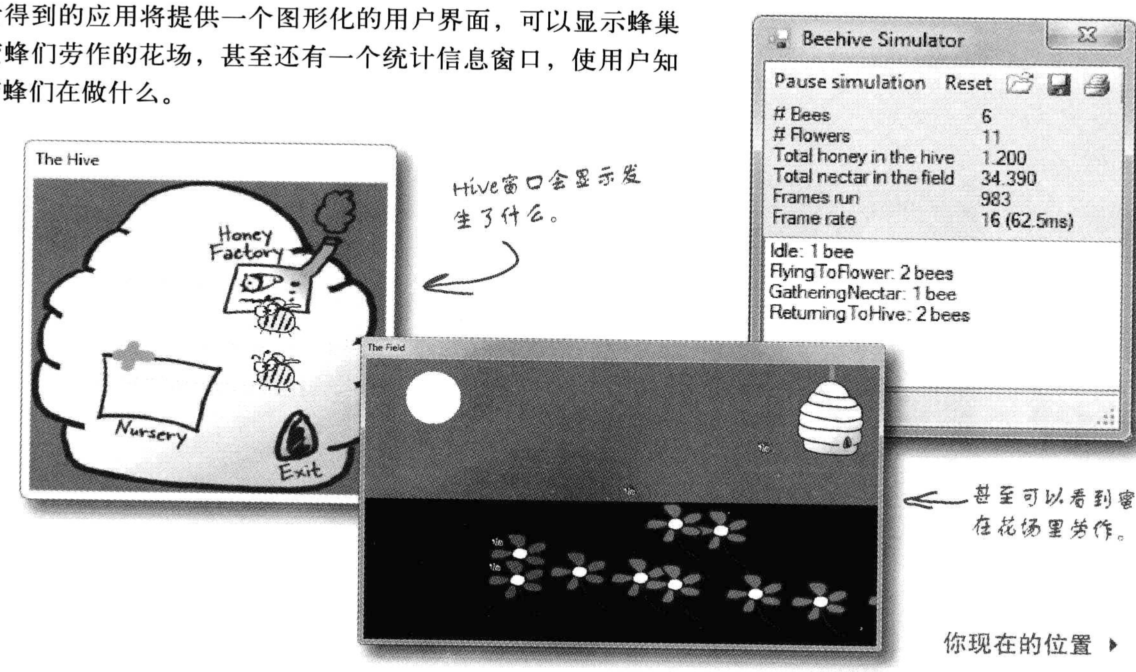

……甚至蜜蜂可以轮班工作。

## 不过现在能做得更好……

不过，从第6章以后，我们又学了不少新知识。所以下面从头开始，基于前面几章的内容再来构建一个动画版的蜂巢模拟系统。最后得到的应用将提供一个图形化的用户界面，可以显示蜂巢和蜜蜂们劳作的花场，甚至还有一个统计信息窗口，使用户知道蜜蜂们在做什么。

这个统计信息窗口使我们能够很详细地查看模拟情况。

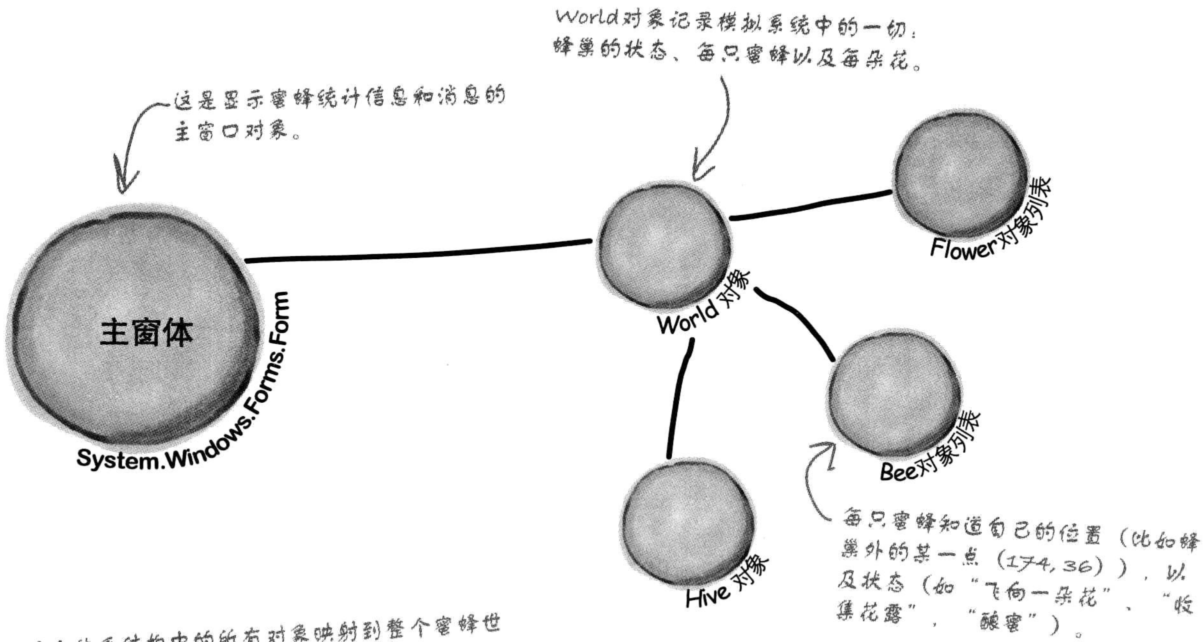
Hive窗口会显示发生了什么。

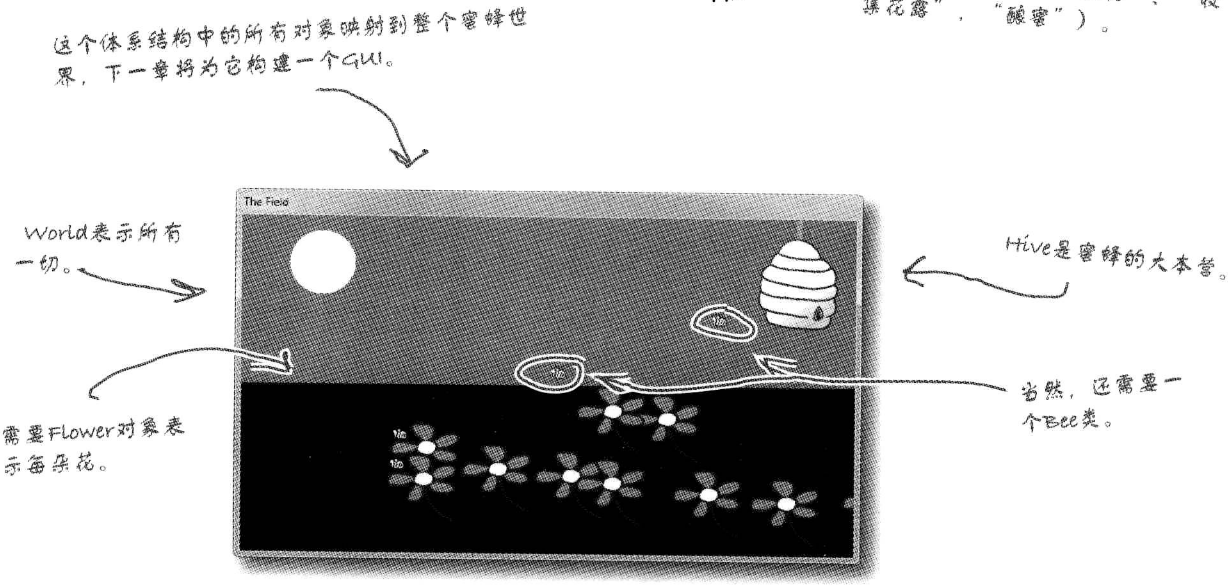



甚至可以看到蜜蜂在花场里劳作。

# 蜂巢模拟系统体系结构

下面是这个蜂巢模拟系统的体系结构。尽管这个模拟系统要控制大量不同的蜜蜂，不过整个对象模型很简单。



# 构建蜂巢模拟系统

当然，之前我们没有构建过这种复杂的应用，所以我们将用两章把有关的各个部分集成起来。在这个过程中，我们还将补充定时器、LINQ和一些图形化技术。

以下是这一章要做的工作（下一章会完成另外一些工作）：

1. **构建一个Flower类，它能成熟、生产花露，最后枯萎、凋谢。**

2. **构建一个Bee类，它有多种不同的状态（从一朵花收集花露，返回蜂巢等），而且根据状态知道要做什么。**

3. **构建一个Hive类，有一个入口、出口，新生蜜蜂的保育室，还有蜂蜜工厂，能够把收集来的花露变为蜂蜜。**

4. **构建一个World类，管理任意指定时刻的蜂巢、花和蜜蜂。**

5. **创建一个主窗体，收集其他类的统计信息，并保证一切正常运转。**

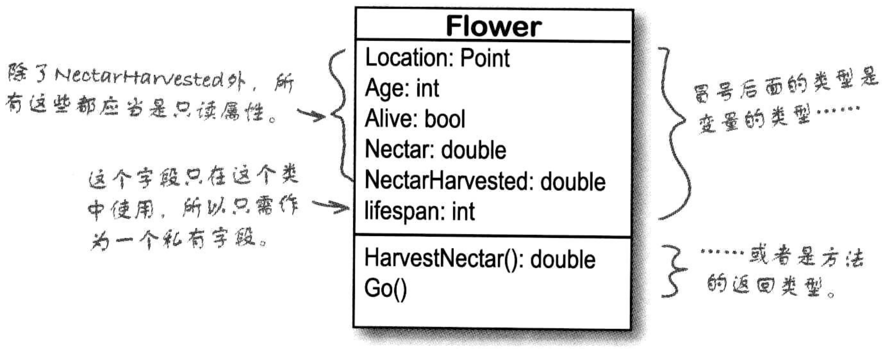
**Exercise**

先来完成一些代码。首先，我们需要一个Flower类。Flower类有一个位置（由Point定义）、年龄和寿命。随着时间推移，花会逐渐变老。当年龄达到它的寿命时，这朵花就会凋谢。你的任务就是具体实现这个Flower类。

类"骨架"只包含它的字段、属性和方法声明，不包括具体实现。

**1 编写Flower的骨架代码。**

下面是`Flower`的类图。编写基本类骨架。`Location`、`Age`、`Alive`、`Nectar`和`NectarHarvested`都是自动属性。`NectarHarvested`可写，其他4个属性只读。现在方法暂时为空；稍后再来实现这些方法。




**2 为这个类增加一些常量。**

对于花需要很多常量。为`Flower`类增加以下6个常量：

- `LifeSpanMin`，花的最短寿命。
- `LifeSpanMax`，花的最长寿命。
- `InitialNectar`，一朵花最初有多少花露。
- `MaxNectar`，一朵花能容纳多少花露。
- `NectarAddedPerTurn`，花逐渐变老时每次增加多少花露。
- `NectarGatheredPerTurn`，一个周期内收集多少花露。

又及，类图中一般不显示常量。

应该能够根据这些常量的值确定它们的类型。花可以生存15000～30000个周期，开始时有1.5个单位的花露。最多可以容纳5个单位的花露。每个周期花会增加0.01个单位的花露，另外一个周期能收集0.3个单位的花露。

由于这个模拟系统将是一个动画系统，我们将逐帧绘制，这里会交替使用"帧"、"周期"和"回合"。

使用Point的所有类文件最前面需要增加using System.Drawing;。

**3 构建构造函数。**

Flower的构造函数取一个Point（指示花的位置）以及Random类的一个实例。应该能使用这些参数来设置花的位置（Location），然后把它的年龄（Age）设置为0，设置这朵花还开着（Alive），并设置其花露量（Nectar）为花的初始花露量。由于还没有收集花露，所以还要适当地设置这个变量（NectarHarvested）。最后，确定这朵花的寿命（lifespan）。下面这行代码应该对你有帮助：

```
lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
```

只有当变量、常量以及Flower构造函数的参数命名正确时，这个代码才能工作。

**4 为HarvestNectar()方法编写代码**

每次调用这个方法时，它要查看每个周期收集的花露是否大于剩余的花露量。如果是，则返回0。否则，应该将这朵花剩余的花露量减去一个周期收集的花露量，并返回收集了多少花露。对了，别忘了把这个量增加到NectarHarvested变量，这个变量会跟踪从这朵花总共收集了多少花露。

提示：在这个方法中我们使用了NectarGatheredPerTurn、nectar和NectarHarvested，不过除此以外，没有使用其他变量或常量。

**5 为Go()方法编写代码。**

这个方法会使花生长。假设每次调用这个方法时，就过去了一个周期，所以要适当地更新花的年龄。还需要查看这个年龄是否大于花的寿命。如果是，那么这朵花凋谢。

假设花还开着，需要增加每朵花在一个周期里新得的花露量。要检查是否达到这朵花所能容纳的最大花露量，不能超过这个上限。

最后的应用是个动画应用，有一些蜜蜂的小图片飞来飞去。每个时间帧都会调用Go()方法，每秒将有多个时间帧。

**答案在下一页……看答案之前先自己试着完成这个代码并编译。**

**Exercise Solution**

你的任务是为这个蜂巢模拟系统构建Flower类。

| Flower |
|---|
| Location: Point<br>Age: int<br>Alive: bool<br>Nectar: double<br>NectarHarvested: double<br>lifespan: int |
| HarvestNectar(): double<br>Go() |

```
class Flower {
    private const int LifeSpanMin = 15000;
    private const int LifeSpanMax = 30000;
    private const double InitialNectar = 1.5;
    private const double MaxNectar = 5.0;
    private const double NectarAddedPerTurn = 0.01;
    private const double NectarGatheredPerTurn = 0.3;
    public Point Location { get; private set; }
    public int Age { get; private set; }
    public bool Alive { get; private set; }
    public double Nectar { get; private set; }
    public double NectarHarvested { get; set; }
    private int lifeSpan;

    public Flower(Point location, Random random) {
        Location = location;
        Age = 0;
        Alive = true;
        Nectar = InitialNectar;
        NectarHarvested = 0;
        lifeSpan = random.Next(LifeSpanMin, LifeSpanMax + 1);
    }

    public double HarvestNectar() {
        if (NectarGatheredPerTurn > Nectar)
            return 0;
        else {
            Nectar -= NectarGatheredPerTurn;
            NectarHarvested += NectarGatheredPerTurn;
            return NectarGatheredPerTurn;
        }
    }

    public void Go() {
        Age++;
        if (Age > lifeSpan)
            Alive = false;
        else {
            Nectar += NectarAddedPerTurn;
            if (Nectar > MaxNectar)
                Nectar = MaxNectar;
        }
    }
}
```

Location, Age, Alive, Alive和Nectar都是只读的自动属性。

其他类应该能访问NectarHarvested。

花的寿命是随机的，所以花的这个字段不会突然改变。

蜜蜂调用HarvestNectar()从花收集花露。蜜蜂一次只能收集一点花露，所以它必须在花旁边来回多次，直到取完花露。

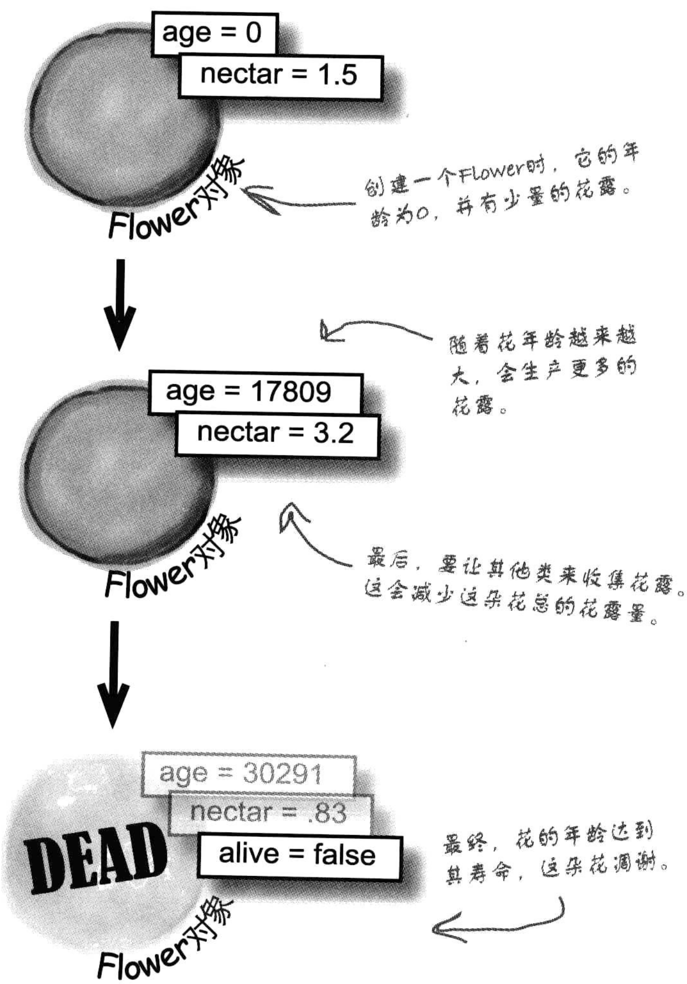
作为这个模拟系统动画的一部分，每个时间帧都会调用Go()方法。这使得每个时间帧花的年龄都稍有增加，随着模拟系统的运行，年龄的增加会逐渐累积。

确保花凋谢后不再增加花露。

**Point位于System.Drawing命名空间，所以一定要在类文件最前面增加using System.Drawing;。**

# 花的生与死

花会经历一个基本的轮回：开花，增加花露，收集花露，最终凋谢。




## there are no Dumb Questions

**问：看起来这个类里没有用到NectarHarvested，我们只是让它递增。这个变量有什么用处？**

**答：** 问得好！在这里我们提前做了一点规划。这个模拟系统最终会一直监视这些花，查看总共收集了多少花露，把数据交给统计窗口显示。所以这里先不要管它，稍后其他类就会用到这个变量。

**问：为什么都是只读自动属性？**

**答：** 还记得第5章吗？我们提到过隐藏私有字段，这通常是一个很好的实践。Flower可以负责维护这些值，所以将它们置为只读。其他对象（如蜜蜂和蜂巢）应该能读取这些属性，但是不能修改。不过要雇，它们只是在类以外是只读的，类内部的代码完全可以访问私有的设置存取方法。

**问：我的代码看起来和你的不一样。我做错了吗？**

**答：** 你的各个方法中的代码顺序可能不同，不过，只要你的代码与我们的功能相同，那就行了。这正体现了封装的另一个方面：每个类的内部细节对于其他类来说并不重要，只要每个类能完成它应该完成的工作就行。

## BRAIN POWER

如果Go()将Flower的年龄增1，而花的寿命在15000～30000之间，这意味着对于每朵花，在它凋谢之前至少要调用15000次Go()方法。这个方法要调用这么多次，你将如何处理？如果有10朵花呢？100朵呢？1000朵呢？

# 现在需要一个Bee类

有了能收集花露的鲜花，下面需要一个Bee类。以下是Bee的基本代码。Bee知道自己的年龄，在不在蜂巢里，以及它能收集多少花露。我们还增加了一个方法，可以让蜜蜂移向一个特定的目标点。

```
class Bee {
  private const double HoneyConsumed = 0.5;
  private const int MoveRate = 3;
  private const double MinimumFlowerNectar = 1.5;
  private const int CareerSpan = 1000;

  public int Age { get; private set; }
  public bool InsideHive { get; private set; }
  public double NectarCollected { get; private set; }

  private Point location;
  public Point Location { get { return location; } }

  private int ID;
  private Flower destinationFlower;

  public Bee(int id, Point location) {
    this.ID = id;
    Age = 0;
    this.location = location;
    InsideHive = true;
    destinationFlower = null;
    NectarCollected = 0;
  }

  public void Go(Random random) {
    Age++;
  }
```

类似于Flower类，需要定义一些特定于蜜蜂的常量。

利用MinimumFlowerNectar，蜜蜂可以确定能从哪些花收集花露。

这里对位置使用了一个后备字段。如果使用自动属性，MoveTowardsLocation()就不能直接设置它的成员（“Location.X -= MoveRate”）。

会为每个蜜蜂指定它自己唯一的ID号。

蜜蜂需要一个ID和一个初始位置。

蜜蜂开始时都在蜂巢里，没有作为目标的花，也没有任何花露。

后面还需要为Go()补充更多的代码，不过，可以以此作为起点。

这里使用了Math.Abs()来计算目标和当前位置之差的绝对值。

```
private bool MoveTowardsLocation(Point destination) {
    if (Math.Abs(destination.X - location.X) <= MoveRate &&
        Math.Abs(destination.Y - location.Y) <= MoveRate)
        return true;

    if (destination.X > location.X)
        location.X += MoveRate;
    else if (destination.X < location.X)
        location.X -= MoveRate;

    if (destination.Y > location.Y)
        location.Y += MoveRate;
    else if (destination.Y < location.Y)
        location.Y -= MoveRate;

    return false;
}
```

这个方法首先得出是否已经在离目标相差MoveRate的范围之内。

如果蜜蜂到达目标，这个方法返回true；否则返回false。

如果还不够接近，再按移动速度向目标移动。

这里返回false，因为还没有到达目标点。需要继续移动。

MoveTowardsLocation()通过修改蜜蜂location字段的X和Y值，将蜜蜂的当前位置向目标移动。如果到达目标位置，则返回true。




蜜蜂可以做很多事情。下面给出一个列表。请创建Bee使用的一个新的enum，名为BeeState。还可以为每个Bee创建一个只读的自动属性，名为CurrentState，用于跟踪蜜蜂的状态。将蜜蜂的初始状态设置为空闲（Idle），并在Go()方法中增加一个switch语句，对应enum中的各个枚举项分别有一个case分支。

| 枚举项 | 含义 |
|---|---|
| Idle | 这个蜜蜂什么也没做 |
| FlyingToFlower | 这个蜜蜂正飞向一朵花 |
| GatheringNectar | 这个蜜蜂正在从一朵花收集花露 |
| ReturningToHive | 这个蜜蜂正飞回蜂巢 |
| MakingHoney | 这个蜜蜂正在酿蜜 |
| Retired | 这个蜜蜂正把翅膀收起来 |




蜜蜂可以做很多事情。下面给出一个列表。请创建Bee使用的一个新的enum，名为BeeState。还可以为每个Bee创建一个只读的自动属性，名为CurrentState，用于跟踪蜜蜂的状态。将蜜蜂的初始状态设置为空闲（Idle），并在Go()方法中增加一个switch语句，对应enum中的各个枚举项分别有一个case分支。

```
enum BeeState {
  Idle,
  FlyingToFlower,
  GatheringNectar,
  ReturningToHive,
  MakingHoney,
  Retired
}

class Bee {
  // constant declarations
  // variable declarations

  public BeeState CurrentState { get; private set; }

  public Bee(int ID, Point initialLocation) {
    this.ID = ID;
    Age = 0;
    location = initialLocation;
    InsideHive = true;
    CurrentState = BeeState.Idle;
    destinationFlower = null;
    NectarCollected = 0;
  }
```

这是包含所有蜜蜂状态的enum。

还需要一个变量来跟踪各个蜜蜂的状态。

蜜蜂开始是空闲的。

记得要在类文件最前面增加using System.Drawing; 吗（因为它使用了Point？）

我们已经填入了一些状态的相应代码。如果你没有理解这些代码也没关系，可以先照单增加。

```
public void Go(Random random) {
    Age++;
    switch (CurrentState) {
     case BeeState.Idle:
       if (Age > CareerSpan) {
         CurrentState = BeeState.Retired;
       } else {
         // What do we do if we're idle?
       }
       break;
     case BeeState.FlyingToFlower:
        // move towards the flower we're heading to
        break;
     case BeeState.GatheringNectar:
       double nectar = destinationFlower.HarvestNectar();
       if (nectar > 0)
         NectarCollected += nectar;
       else
         CurrentState = BeeState.ReturningToHive;
       break;
     case BeeState.ReturningToHive:
       if (!InsideHive) {
         // move towards the hive
       } else {
         // what do we do if we're inside the hive?
       } break;
     case BeeState.MakingHoney:
       if (NectarCollected < 0.5) {
         NectarCollected = 0;
         CurrentState = BeeState.Idle;
       } else {
         // once we have a Hive, we'll turn the nectar into honey
       }
       break;
     case BeeState.Retired:
       // Do nothing! We're retired!
       break;
    }
  }
}
```

这是处理各个蜜蜂状态的switch()语句。

如果这个蜜蜂的年龄已经达到它的寿命，这只蜜蜂就会退休。不过在它退休前会先完成当前的工作。

稍后会增加这里的代码。

在这里从前面编写的花收集花露……。

……如果还有剩余，把它增加到已收集的花露。

所有状态都应当涵盖。

……不过，如果没有剩下花露，蜜蜂就飞回蜂巢。

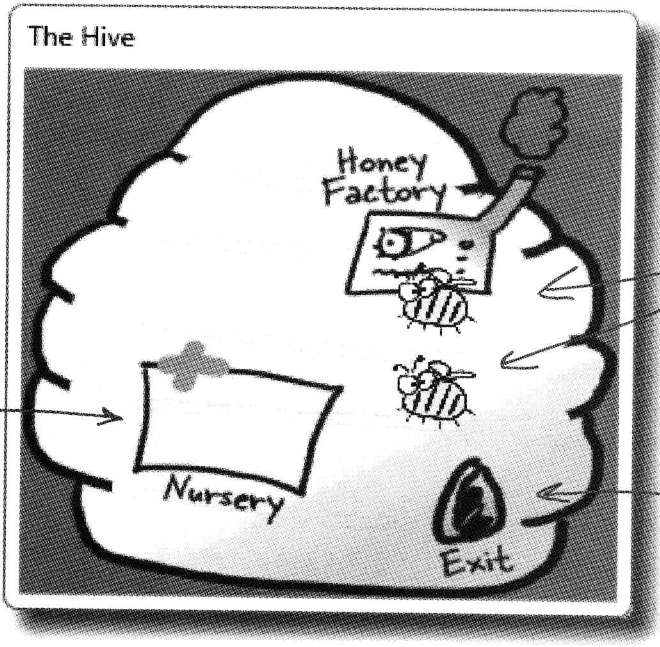
取决于是否已经在蜂巢中，返回蜂巢的具体做法有所不同。

蜜蜂一次向蜂蜜工厂增加半个单位的花露。如果没有收集到足够的花露（不到0.5个单位），工厂就不能使用它收集的花露，所以蜜蜂只能将收集的花露丢掉。

# P.A.H.B.(程序员与无家可归的蜜蜂)

我们已经有了蜜蜂，还有装满花露的花。下面需要编写一些代码使蜜蜂能够收集花露，不过在此之前，首先需要考虑应该在哪里创建蜜蜂？它们把所有这些花露送到什么地方？这就需要一个Hive类。

不过，蜂巢并不只是让蜜蜂有家可回。蜂巢里有很多位置，它们位于这个世界中不同的点上。蜂巢里有一个入口和出口，另外还有一个保育室来培育更多的蜜蜂，还有一个蜂蜜工厂把花露酿成蜂蜜。




每个位置都不同，蜜蜂可以从一个位置移到另一个位置，就像可以从蜂巢飞向花一样。

新蜜蜂在蜂巢的保育室里创建，并从这里出发。

蜜蜂从入口进入，从出口离开，很有秩序。

# 蜂巢靠蜂蜜运转

蜂巢还有一个重要作用，它要跟踪储存了多少蜂蜜。蜂巢要靠蜂蜜才能正常运转，如果需要创建新的蜜蜂，这也要消耗蜂蜜。在蜂巢的最上面，蜂蜜工厂必须将蜜蜂们收集的花露酿成蜂蜜。一个单位的花露可以酿出0.25个单位的蜂蜜。

先考虑几秒钟……随时间推移，蜂巢使用蜂蜜来运转，并创建更多的蜜蜂。与此同时，其他蜜蜂会带回花露，这些花露又会变成蜂蜜，这就能让蜂巢维持更长时间。要由你来完成模拟系统中的这部分代码对所有这些建模（我们会提供一些帮助）。

**Exercise**

要由你来编写Hive的代码。

| Hive |
| --- |
| Honey: double<br>locations: Dictionary<string, Point><br>beeCount: int |
| InitializeLocations()<br>AddHoney(Nectar: double): bool<br>ConsumeHoney(amount: double): bool<br>AddBee(random: Random)<br>Go(random: Random)<br>GetLocation(location: string): Point |

1. **编写Hive的骨架代码。**
与Flower类一样，首先要编写Hive的一个基本骨架。右边给出了它的类图。Honey要作为一个只读的自动属性，locations应当是私有的，beeCount只能在内部使用，所以这可以是一个私有字段。

2. **为Hive定义常量。**
需要一些常量分别表示初始蜂蜜数（6），蜂巢最初的蜂蜜量(3.2)，蜂巢能储存的最大蜂蜜量(15)，每单位花露能生产的蜂蜜量 (.25),蜜蜂最大数目(8)以及蜂巢中要繁殖新蜜蜂所需的最低蜂蜜量(4)。

必须为各个常量确定有意义的名字和合适的类型。确定类型时，不要只是考虑初始值，还要考虑这些常量将使用的值。double最好与其他double搭配，int与int搭配。

3. **编写代码处理Locations。**
首先编写GetLocation()方法。它取一个string参数，在Locations字典中查找这个串，并返回与之关联的点（Point）。如果没有找到这个串，则抛出一个ArgumentException。

接下来编写InitializeLocations()方法。这个方法应当在蜂巢中创建以下位置：

- Entranc，位于(600, 100)
- Nursery，位于(95, 174)
- HoneyFactory，位于(157, 98)
- Exit，位于(194, 213)

这些位置分别映射到蜂巢所在2D空间中的一个位置。后面还要确保模拟系统使蜂巢覆盖所有这些点。

在这个模拟系统中，我们只是假设有一个蜂巢，并有固定的位置点。如果希望有多个蜂巢，可以相对于蜂巢来设置位置点，而不是相对于整个世界。

4. **构建Hive构造函数。**
构造蜂巢时，应当将其蜂蜜量设置为所有蜂巢的初始蜂蜜量。需要建立蜂巢中的位置，还要创建Random的一个新实例。然后，对于在蜂巢出生的每个蜜蜂分别调用一次AddBee()，并传入刚才创建的Random实例。

AddBee()需要一个Random对象，因为它要向Nursery位置增加一个随机值，这样一来，蜜蜂就不会彼此重叠了。




你的任务是构建Hive类。

一定要增加一行“using System.Drawing;”，因为以下代码中使用了Point。

你的常量可以有不同的名字。具体用什么常量名没有关系，只要在其余的代码中保持一致就可以。

这里置MaximumHoney为double类型，因为蜂蜜量可以取值为InitialHoney(3.2)到这个值(MaximumHoney)之间的某个值。由于InitialHoney必须是一个double，所以最好把MaximumHoney也置为一个double。

```
class Hive {
  private const int InitialBees = 6;
  private const double InitialHoney = 3.2;
  private const double MaximumHoney = 15.0;
  private const double NectarHoneyRatio = .25;
  private const double MinimumHoneyForCreatingBees = 4.0;
  private const int MaximumBees = 8;

  private Dictionary<string, Point> locations;
  private int beeCount = 0;

  public double Honey { get; private set; }

  private void InitializeLocations() {
    locations = new Dictionary<string, Point>();
    locations.Add("Entrance", new Point(600, 100));
    locations.Add("Nursery", new Point(95, 174));
    locations.Add("HoneyFactory", new Point(157, 98));
    locations.Add("Exit", new Point(194, 213));
  }

  public Point GetLocation(string location) {
    if (locations.Keys.Contains(location))
       return locations[location];
    else
       throw new ArgumentException("Unknown location: " + location);
 }

 public Hive() {
   Honey = InitialHoney;
   InitializeLocations();
   Random random = new Random();
   for (int i = 0; i < InitialBees; i++)
     AddBee(random);
 }

 public bool AddHoney(double nectar) { return true; }
 public bool ConsumeHoney(double amount) { return true; }
 private void AddBee(Random random) { }
 public void Go(Random random) { }
}
```

还记得字典吗？这个字典存储一个位置，并以一个string值作为键。

不要忘记创建Dictionary的一个新实例，否则将不能正常工作。

这个方法的其余部分很简单。

这个方法可以防止其他类处理这个位置字典，而且不能非法进行修改。这正是封装的一个很好的例子。

应该对蜂巢中初始的每个蜜蜂分别调用一次AddBee()。

还没有为这些方法编写代码，不过应该创建一些空方法作为占位符。

也可以在尚未实现的任何方法中抛出一个NotImplementedException。这是一种很好的方法，可以跟踪还需要构建哪些代码。

用这种方式构建代码不奇怪吗？我们的蜜蜂不了解花，而且蜂巢里满是空方法声明。现在什么也做不了，不是吗？



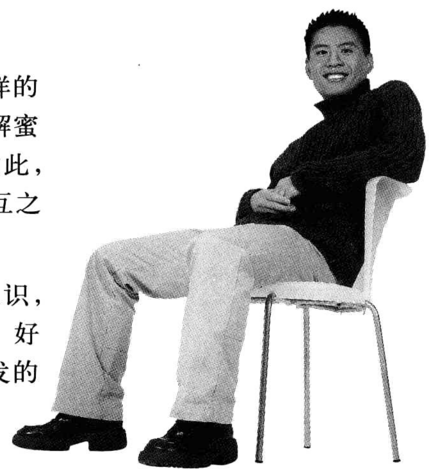
**实际代码是一点一点构建的。**

如果能一次写完一个类的所有代码，完成编译，通过测试，然后放在一边，再开始编写另一个类，这样该多好。遗憾的是，这几乎是不可能的。

大多数情况下，你都会像这一章这样编写代码：一点一点地编写。我们也可以直接构建完整的Flower类，但是等到编写Bee时，会发现还是需要对Flower类再做一些工作（告诉它对于各种状态需要做什么处理）。

现在，Hive中有很多空方法需要填写。另外，还没有将任何Bee与Hive绑定。而且，要对所有这些对象调用成千上万次Go()方法，这里还存在问题……

不过我们还没有真正开始，还没有把这些类集成在一起！首先会确定体系结构，然后才开始构建。



**先设计，再构建。**

开始创建这个工程时，我们很清楚要构建一个怎样的应用，这是一个蜂巢模拟系统。而且我们非常了解蜜蜂、花、蜂巢和世界相互之间如何协作。正因如此，我们先从体系结构开始，它能告诉我们各个类相互之间如何作用。然后再转向各个类，分别进行设计。

开始构建之前如果对所要构建的应用有清楚的认识，你的工程会进行得更加顺利。这一点看起来很简单，好像是人所共知的常识。但正是这一点会让最终开发的应用大相径庭。

# 填写Hive类

再回到Hive类，填写原来没有实现的那些方法：

```
class Hive {
  // constant declarations
  // variable declarations

  // InitializeLocations()
  // GetLocation()
  // Hive constructor

  public bool AddHoney(double nectar) {
    double honeyToAdd = nectar * NectarHoneyRatio;
    if (honeyToAdd + Honey > MaximumHoney)
        return false;
    Honey += honeyToAdd;
    return true;
  }
  public bool ConsumeHoney(double amount) {
    if (amount > Honey)
      return false;
    else {
      Honey -= amount;
      return true;
    }
  }
  private void AddBee(Random random) {
    beeCount++;
    int r1 = random.Next(100) - 50;
    int r2 = random.Next(100) - 50;
    Point startPoint = new Point(locations["Nursery"].X + r1,
                                 locations["Nursery"].Y + r2);
    Bee newBee = new Bee(beeCount, startPoint);
    // Once we have a system, we need to add this bee to the system
  }
  public void Go(Random random) { }
}
```

首先，确定这些花露能酿出多少蜂蜜……

……然后查看蜂巢里有没有足够的空间存放这么多蜂蜜。

如果有足够的空间，把这些蜂蜜增加到蜂巢。

这个方法取一定量的蜂蜜，并消耗这些蜂蜜，从蜂巢的库存将其减去。

如果蜂巢里没有足够的蜂蜜，不能满足要求，则返回false。

如果有足够的蜂蜜，从蜂巢的库存将其减去，并返回true。

这是私有的……只有Hive实例能创建蜜蜂。

这会在距保育室位置X和Y方向50个单位以内的某个位置创建一个点。

在指定的位置增加一个新蜜蜂。

稍后将完成AddBee()，并填写Go()方法……

# 蜂巢的Go()方法

我们已经为Flower编写了一个Go()方法，另外为Bee也编写了一个Go()方法（尽管还需要补充一些代码）。以下是Hive的Go()方法：

```
public void Go(Random random) {
  if (Honey > MinimumHoneyForCreatingBees)
    AddBee(random);
}
```

唯一的限制是（至少对现在来讲）：蜂巢必须有足够的蜂蜜才能创建更多蜜蜂。

传入Go()的Random实例会发送到AddBee()方法。

遗憾的是，这并不现实。大多数情况下，在一个忙碌的蜂巢里，蜂王没有时间来创建更多蜜蜂。我们没有QueenBee类，不过假设有足够的蜂蜜可以创建蜜蜂，那么真正创建新蜜蜂的几率是10%。对此可以如下建模：

```
public void Go(Random random) {
  if (Honey > MinimumHoneyForCreatingBees
        && random.Next(10) == 1) {
    AddBee(random);
  }
}
```

这是一种模拟以1/10的几率创建蜜蜂的简单方法。这会得到一个0～9之间的随机数。如果这个数为1，则创建蜜蜂。

## there are no Dumb Questions

**问：** 这么说来，蜂巢可以创建无限个蜜蜂了？

**答：** 目前确实可以，或者至少上限很大，不过你的怀疑是有道理的，这并不现实。后面我们还会讨论这个问题，并增加一个限制，要求某一时刻这个模拟世界中只能存在一定数目的蜜蜂。

**问：** 难道不能把Random的这个实例赋给类的一个属性吗？而不是把它传递给AddBee()。

**答：** 当然可以。这样一来，AddBee就可以使用这个属性，而不是使用传入的一个参数。对此并没有惟一的正确答案；采用哪种方式完全由你决定。

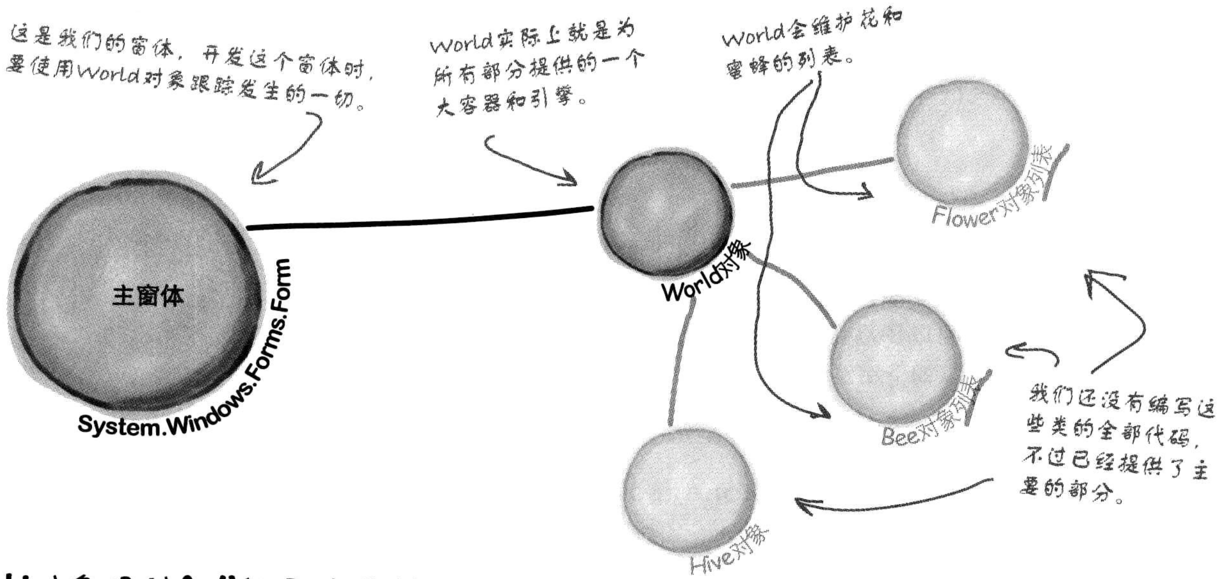
这里没有考虑这种做法，原因之一是，这样就能保存Random种子——这样一来，如果愿意，以后可以再次运行某一次特定的模拟……如果你想这样做！

**问：** 我还是不清楚所有这些Go()方法是如何调用的。

**答：** 没关系，我们刚刚谈到这个内容。不过，首先还需要另外一个对象：World类，它将记录蜂巢里发生的一切，跟踪所有蜜蜂，甚至记录花的状况。

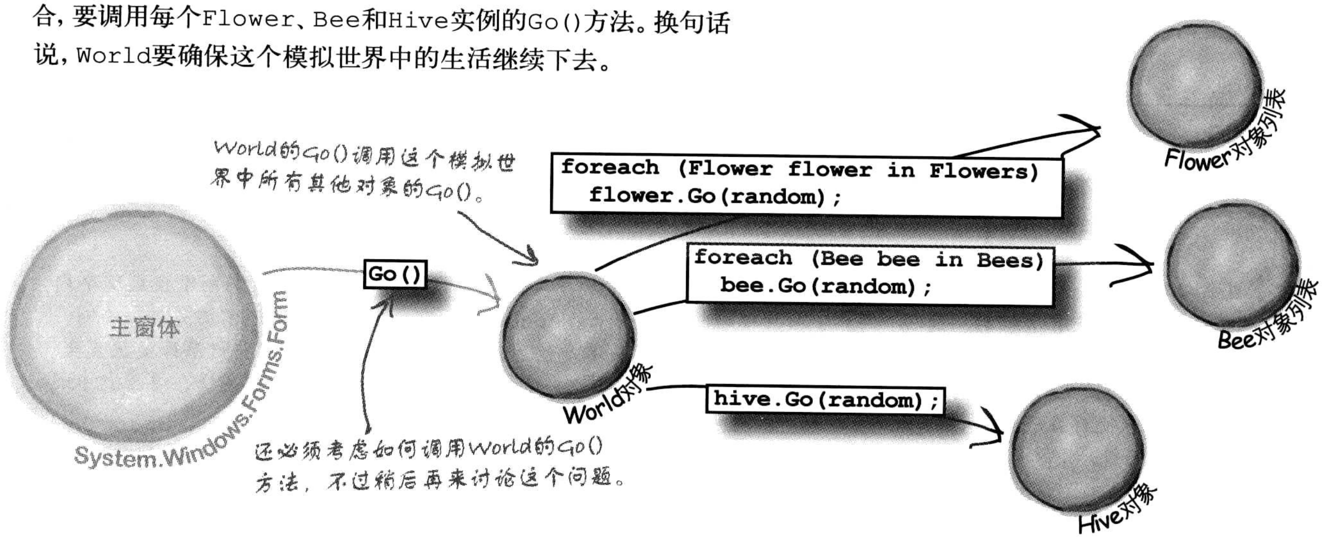
# 准备构建World

有了Hive，Bee和Flower类，最后终于可以构建World类了。World处理模拟系统中各部分之间的协调：跟踪所有蜜蜂，告诉蜂巢是否有容纳更多蜜蜂的空间，找到花的位置等。




## World对象通过Go()记录发生的一切

World对象最重要的任务之一是，对于模拟系统中的每一回合，要调用每个Flower、Bee和Hive实例的Go()方法。换句话说，World要确保这个模拟世界中的生活继续下去。

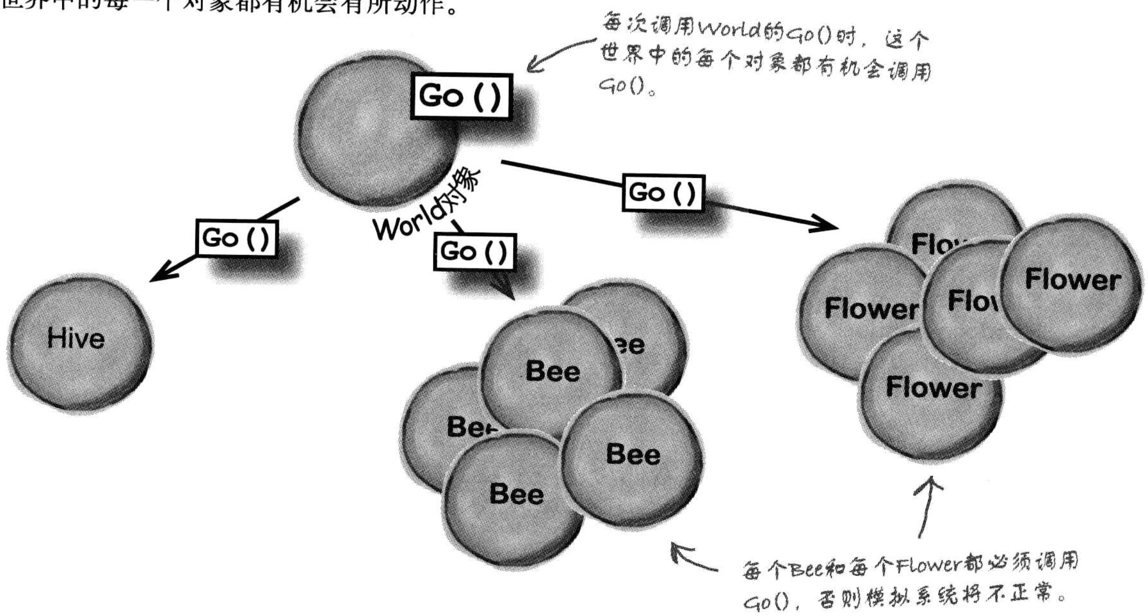


# 我们要构建一个基于回合的系统

在这个模拟系统中，每一回合（或每个周期）都会运行各个对象的Go()方法。这里的一回合是指任意的一段时间……例如，一回合可能是每隔10秒；或者每隔60秒，或者每隔10分钟。

每一“回合”将绘制为一个动画帧，所以每一回合World只需要改变一点点。

重要的是，一个回合会影响这个世界中的每一个对象。蜂巢的年龄会增大“一回合”，并查看是否需要增加更多蜜蜂。接下来各个蜜蜂会向其目标移动一段很小的距离，或者完成一个很小的动作，而且年龄会增加。另外每朵花会制造一点点花露，同样地，年龄也会增加。这就是World所要做的：它确保每次调用其Go()方法时，这个世界中的每一个对象都有机会有所动作。




## Sharpen your pencil

在这个模拟系统中我们一直在使用4大面向对象原则之一：封装（可以翻回第5章复习有关内容）。查看目前为止开发的代码，看看在所构建的各个类中能不能找出两个封装的例子。

**Hive**

1. ........................................

........................................

2. ........................................

........................................

**Bee**

1. ........................................

........................................

2. ........................................

........................................

**Flower**

1. ........................................

........................................

2. ........................................

........................................

# World的代码

> **封装警告!**
>
> 仔细查看public Hive、Bees和Flowers字段。另一个类可能会无意中将其中任何字段重置为null，这可能会导致严重的问题!你能想出一种方法使用属性或方法更好地完成封装吗?

World类实际上是这个模拟系统中比较简单的一个类。以下代码可以作为起点。不过如果你仔细查看，会注意到这里少了一些东西（稍后就会增加）：

```
using System.Drawing;
class World {
    private const double NectarHarvestedPerNewFlower = 50.0;
    private const int FieldMinX = 15;
    private const int FieldMinY = 177;
    private const int FieldMaxX = 690;
    private const int FieldMaxY = 290;

    public Hive Hive;
    public List<Bee> Bees;
    public List<Flower> Flowers;

    public World() {
        Bees = new List<Bee>();
        Flowers = new List<Flower>();
        Random random = new Random();
        for (int i = 0; i < 10; i++)
            AddFlower(random);
    }

    public void Go(Random random) {
        Hive.Go(random);

        for (int i = Bees.Count - 1; i >= 0; i--) {
            Bee bee = Bees[i];
            bee.Go(random);
            if (bee.CurrentState == BeeState.Retired)
                Bees.Remove(bee);
        }

        double totalNectarHarvested = 0;
        for (int i = Flowers.Count - 1; i >= 0; i--) {
            Flower flower = Flowers[i];
            flower.Go();
            totalNectarHarvested += flower.NectarHarvested;
            if (!flower.Alive)
                Flowers.Remove(flower);
        }
```

这些字段定义了花场的边界，也就是花生长的地方。

每个世界都有一个蜂巢，一个蜜蜂列表以及一个花列表。

创建一个新世界时，要初始化各个列表，创建一个新蜂巢，然后增加最初的10朵花。

这很容易……只要告诉Hive调用Go()，并传入一个Random实例。

循环处理当前的所有蜜蜂，并告诉它们调用Go()。

如果一个蜜蜂退休了，需要把它从这个世界取出。

循环处理每朵花，告诉它们调用Go()。

还需要跟踪这一回合收集了多少花蜜，可以把从每朵花收集的花蜜累加起来得到。

类似于蜜蜂的处理，要把这一回合凋谢的花删除。

## Sharpen your pencil Solution

在这个模拟系统中我们一直在使用4大面向对象原则之一：封装（可以翻回第5章复习有关内容）。查看目前为止开发的代码，看看在所构建的各个类中能不能找出两个封装的例子。

这些是我们找到的例子。你能找出其他的例子吗？

**Hive**

1. 蜂巢的Locations字典是私有的。
2. 它为蜜蜂提供了一个增加蜂蜜的方法。

**Bee**

1. 蜜蜂的location是只读的。
2. 年龄也只读，所以其他类不能写这些属性。

**Flower**

1. 花提供了一个收集花露的方法。
2. 并置其alive布尔字段为私有字段。

```
    if (totalNectarHarvested > NectarHarvestedPerNewFlower) {
        foreach (Flower flower in Flowers)
            flower.NectarHarvested = 0;
        AddFlower(random);
    }
}

private void AddFlower(Random random)
{
    Point location = new Point(random.Next(FieldMinX, FieldMaxX),
                               random.Next(FieldMinY, FieldMaxY));
    Flower newFlower = new Flower(location, random);
    Flowers.Add(newFlower);
}
}
```

蜜蜂收集花露时会对花授粉。一旦从花收集了足够的花露，也就完成了必要的授粉，足以让世界增加一朵新的花。

如果花场里有足够的花露，世界就会增加一朵新的花。

这会给出花场中的一个随机的位置……

……然后在这个位置增加一朵新的花。

## there are no Dumb Questions

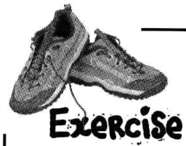
**问：** 为什么不使用foreach循环来删除凋谢的花和退休的蜜蜂呢？

**答：** 因为如果一个foreach循环正在迭代处理一个集合，就无法在这个foreach循环内部删除集合中的项。否则，.NET会抛出一个异常。

**问：** 那好，那为什么这些for循环都从列表的最后一个元素开始，倒数到0呢？

**答：** 因为每个循环都需要保持列表的序号。假设从一个包含5朵花的列表最前面开始处理，循环发现中间的某个花凋谢了，如果它删除索引#3的花，现在列表中只有4朵花，索引#3处有一朵新的花，这朵花最后会被跳过，因为下一次循环会查看索引#4的花。

如果循环从最后开始，移到空槽的花已经由循环处理过，所以不会漏掉任何花。




已经有了4个核心类，下面做一些工作把它们集成起来。遵循以下步骤，另外要保证你的Bee、Hive、Flower和World类能正常工作。不过要注意，完成集成之前，很多地方需要修改，几乎每个类中都必须做一些改动。

**❶ 更新Bee，取一个Hive和World引用。**

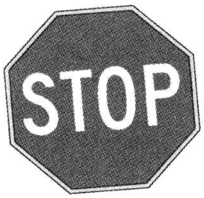
既然已经有了一个`Hive`类和一个`World`类，`Bee`对象需要知道这两个类。更新Bee的代码，在构造函数中取蜜蜂所在蜂巢和这个世界的引用，并保存这两个引用以备以后使用。

**❷ 更新Hive，取一个World引用。**

`Bee`要知道它所在的蜂巢，与此类似，`Hive`也需要知道它所在的世界。更新`Hive`，在构造函数中取一个`World`引用，并保存这个引用。还要更新`Hive`中创建新蜜蜂的代码，为Bee传入它自己（`Hive`）以及`World`的引用。

**❸ 更新World，创建新Hive时传入自己的一个引用。**

更新`World`类，创建新`Hive`时传入它自己（World）的一个引用。




**停！到目前为止，所有代码应该都能编译。如果不能，请仔细检查，修正了所有错误之后再继续。**

**❹ 对Hive能创建的蜜蜂数目加一个上限。**

`Hive`类有一个`MaximumBees`常量，确定了`Hive`能支持多少个蜜蜂（蜂巢内外的蜜蜂都算在内）。由于`Hive`能访问`World`，所以应该能够施加这个限制。

提示：注意创建或增加蜜蜂的代码。Hive中有两处与此相关的代码，所以要务必当心。

**❺ Hive创建蜜蜂时，要让World知道。**

`World`类使用一个`List`跟踪现有的所有蜜蜂。`Hive`创建一个新`Bee`时，要保证这个`Bee`增加到`World`维护的列表上。

## there are no Dumb Questions

**问：为什么在Hive类的GetLocation()方法中抛出一个异常？**

**答：** 因为传入参数的数据不合适时，我们需要一种方法来处理这种情况。蜂巢有一些特定的位置，但是向GetLocations()的参数传入的值可能是任意的串。如果程序中存在bug，导致将一个非法的串作为参数发送给这个方法（如一个空串，或者位置字典中没有的一个位置名），会发生什么情况？这个方法会返回什么结果？

如果得到一个非法参数，而且不太清楚如何处理，抛出一个ArgumentException异常往往是一个很好的想法。GetLocation()方法这这么做的：

```
throw new ArgumentException(
      "Unknown location: " + location);
```

这个语句导致Hive类抛出一个ArgumentException，并提供一个消息“Unknown location:”，其中包含它无法找到的位置。

这很有用，因为一旦将不合适的位置参数传入方法，就会立即提醒你。另外通过在异常消息中包含这个参数，你就能得到一些有价值的信息，将有助于调试这个问题。

**问：既然不打算画出这些位置，何必把所有位置存储在一个Point对象中呢？**

**答：** 每个蜜蜂都有一个位置，而不论你是否在屏幕上的这个位置真正画出蜜蜂。Bee对象的任务就是跟踪蜜蜂在这个世界中的位置。每次调用它的Go()方法时，都需要朝着它的目标移动很小的一段距离。尽管我们现在还不会画出蜜蜂的图片，但是仍然需要记录它在蜂巢里还是在花场飞舞，因为蜜蜂需要知道自己是否已经到达目标。

**问：那为什么使用Point存储位置，为什么不用其他类型呢？Point难道不是专门用于绘图吗？**

**答：** 对，所有可视化控件都使用Point表示其Location属性。不过，尽管.NET这样大量使用Point，但这并不意味着它不能用来跟踪记录我们的位置。不错，我们完全可以创建自己的BeeLocation类，其中包含X和Y整数字段。但是既然C#和.NET已经免费提供了Point，又何必炒冷饭呢！

**如果一个现有的类基本上能完成你需要的所有工作，可以调整或扩展这个类，这往往比完全从头开始编写全新的类更简单。**




已经有了4个核心类，下面做一些工作把它们集成起来。遵循以下步骤，另外要保证你的Bee、Hive、Flower和World类能正常工作。需要做以下修改。

**1** **更新Bee，取一个Hive和World引用。**

既然已经有了一个Hive类和一个World类，Bee对象需要知道这两个类。更新Bee的代码，在构造函数中取蜜蜂所在蜂巢和这个世界的引用，并保存这两个引用以备以后使用。

```
class Bee {
  // existing constant declarations
  // existing variable declarations
  private World world;
  private Hive hive;

  public Bee(int ID, Point InitialLocation, World world, Hive hive) {
    // existing code
    this.world = world;
    this.hive = hive;
  }
}
```

这相当简单……取这些引用，并赋至私有字段。

**2** **更新Hive，取一个World引用。**

Bee要知道它所在的蜂巢，与此类似，Hive也需要知道它所在的世界。更新Hive，在构造函数中取一个World引用，并保存这个引用。还要更新Hive中创建新蜜蜂的代码，为Bee传入它自己（Hive）以及World的引用。

```
class Hive {
  private World world;

  public Hive(World world) {
    this.world = world;
    // existing code
  }
  public void AddBee(Random random) {
    // other bee creation code
    Bee newBee = new Bee(beeCount, startPoint, world, this);
  }
}
```

更多基本代码……取得引用，设置私有字段。要先指定world，因为构造函数中的其余代码需要用到它。

现在，创建新蜜蜂需要world和hive引用。

**如果运行有问题，可以从http://www.headfirstlabs.com/books/hfcsharp/下载这个练习（和所有其他练习）的代码。**

3 **对Hive能创建的蜜蜂数目加一个上限。**

Hive类有一个MaximumBees常量，确定了Hive能支持多少个蜜蜂（蜂巢内外的蜜蜂都算在内）。由于Hive能访问World，所以应该能够施加这个限制。

```
public void Go(Random random) {
  if (world.Bees.Count < MaximumBees
      && Honey > MinimumHoneyForCreatingBees
      && random.Next(10) == 1) {
    AddBee(random);
  }
}
```

可以使用World对象查看总共有多少个蜜蜂，并与蜂巢所能容纳的最大蜜蜂数比较。

先完成这个比较。如果没有更多空间放置蜜蜂，也就没有必要去检查是否有足够的蜂蜜来创建蜜蜂。

4 **Hive创建蜜蜂时，要让World知道。**

World类会跟踪现有的所有蜜蜂。Hive创建一个新Bee时，要保证这个Bee增加到World维护的总列表上。

```
private void AddBee(Random random) {
  beeCount++;
  // Calculate the starting point
  Point startPoint = // start the near the nursery
  Bee newBee = new Bee(beeCount, startPoint, world, this);
  world.Bees.Add(newBee);
}
```

向World维护的蜜蜂列表增加蜜蜂。

这正是Hive类需要一个World引用的一个原因。

5 **更新World，创建新Hive时传入自己的一个引用。**

更新World类，创建新Hive时传入它自己（World）的一个引用。

```
public World() {
  Bees = new List<Bee>();
  Flowers = new List<Flower>();
  Hive = new Hive(this);
  Random random = new Random();
    for (int i = 0; i < 10; i++)
      AddFlower(random);
}
```

向Hive传入当前World的引用。

# 提供蜜蜂的行为

目前这些类中还缺少非常重要的一段代码，就是Bee的Go()方法。前面针对一些状态编写了代码，但是还有许多状态没有相应的代码（Idle还不完全，FlyingToFlower没有实现，另外还缺少MakingHoney的部分代码）。

下面就来完成所有这些不完全的状态：

```
public void Go(Random random) {
    Age++;
    switch (CurrentState) {
        case BeeState.Idle:
            if (Age > CareerSpan) {
                CurrentState = BeeState.Retired;
            } else if (world.Flowers.Count > 0
                 && hive.ConsumeHoney(HoneyConsumed)) {
                Flower flower =
                  world.Flowers[random.Next(world.Flowers.Count)];
                if (flower.Nectar >= MinimumFlowerNectar && flower.Alive) {
                    destinationFlower = flower;
                    CurrentState = BeeState.FlyingToFlower;
                }
            }
            break;
        case BeeState.FlyingToFlower:
            if (!world.Flowers.Contains(destinationFlower))
                CurrentState = BeeState.ReturningToHive;
            else if (InsideHive) {
                if (MoveTowardsLocation(hive.GetLocation("Exit"))) {
                    InsideHive = false;
                    location = hive.GetLocation("Entrance");
                }
            }
            else
                if (MoveTowardsLocation(destinationFlower.Location))
                    CurrentState = BeeState.GatheringNectar;
            break;
        case BeeState.GatheringNectar:
            double nectar = destinationFlower.HarvestNectar();
            if (nectar > 0)
                NectarCollected += nectar;
            else
                CurrentState = BeeState.ReturningToHive;
            break;
```

如果空闲，希望寻找另一朵花来收集花露。

查看是否还有花，然后消耗足够的蜂蜜后继续处理。否则什么也不做。

需要另外一朵有花露而且活着的花。

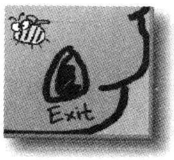
如果找到，则飞向这朵花。

确保在飞向这朵花的过程中它不会凋谢。

正是因为这里需要访问蜂巢，所以要向Bee构造函数传递蜂巢的一个引用。

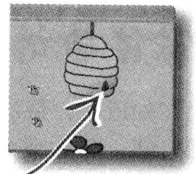
如果到达出口，则飞出蜂巢，更新位置。因为现在在花场窗体上，所以应该飞到接近入口的位置。

如果飞出了蜂巢，而且花还活着，则飞向这朵花，并开始收集花露。




这是出口。蜂巢存储“Exit”位置时，它对应于Hive窗体上显示出口图片的那个点。

这是入口。蜜蜂飞回蜂巢时，它们会飞向花场窗体上的蜂巢入口。



正因如此，位置字典要为出口和入口存储两个单独的位置：“Exit”和“Entrance”。

```
    case BeeState.ReturningToHive:
        if (!InsideHive) {
            if (MoveTowardsLocation(hive.GetLocation("Entrance"))) {
                InsideHive = true;
                location = hive.GetLocation("Exit");
            }
        }
        else
            if (MoveTowardsLocation(hive.GetLocation("HoneyFactory")))
                CurrentState = BeeState.MakingHoney;
        break;
    case BeeState.MakingHoney:
        if (NectarCollected < 0.5) {
            NectarCollected = 0;
            CurrentState = BeeState.Idle;
        }
        else
            if (hive.AddHoney(0.5))
                NectarCollected -= 0.5;
            else
                NectarCollected = 0;
        break;
    case BeeState.Retired:
        // Do nothing! We're retired!
        break;
  }
}
```

如果到达蜂巢，更新位置，并调整insideHive状态。

如果已经在蜂巢里，则飞向蜂蜜工厂。

尝试把这些花露交给蜂巢。

如果蜂巢可以使用这些花露酿造蜂蜜……

……则将这些花露从蜜蜂处删除。

如果蜂巢已满，AddHoney()会返回false，所以蜜蜂只是倒掉剩下的花露，飞出去执行另一个任务。

一旦蜜蜂退休，就只能等待，直到Hive将它从列表中删除。然后它就可以去迈阿密度假了！

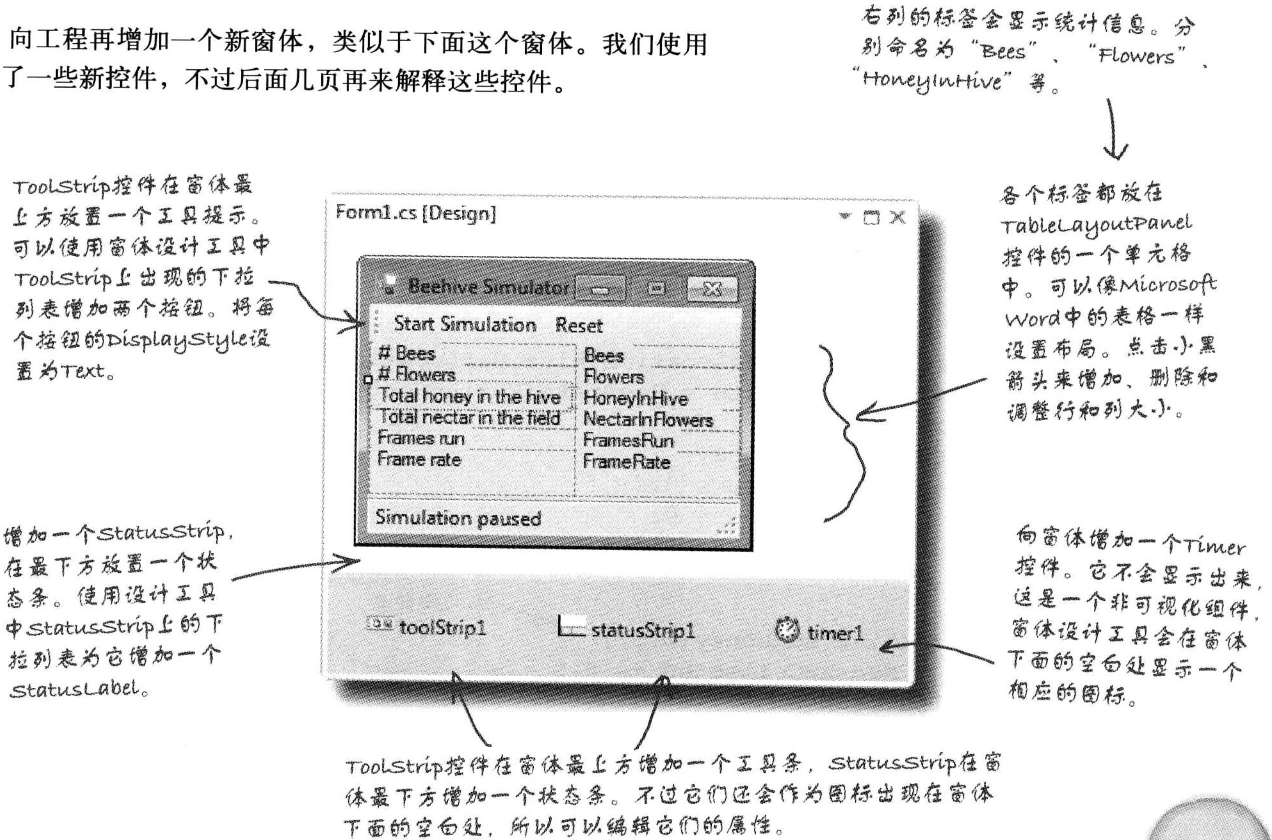
## BRAIN POWER

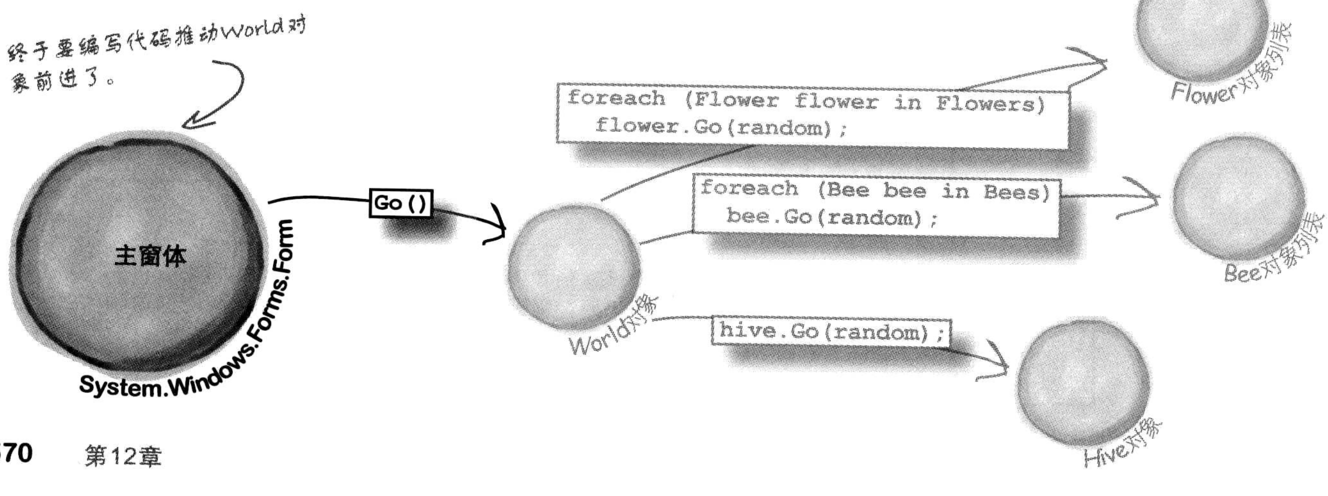
假设希望更新这个模拟系统，从蜂巢飞到花需要两个回合，另外从花飞回蜂巢也需要两个回合。如果不编写任何新的代码，需要修改哪些类的哪些方法才能增加这种新行为？

# 主窗体告诉世界调用Go()推进

你现在知道了，每次世界调用其Go()方法时都会向前推进一个时间帧。那么谁来调用这个Go()方法呢？当然是主窗体。终于该它出场了。

向工程再增加一个新窗体，类似于下面这个窗体。我们使用了一些新控件，不过后面几页再来解释这些控件。



# 可以使用World得到统计信息

现在我们希望更新所有这些控件。不过，不需要分别建立各个控件的点击事件处理程序，相反，我们只使用一个方法，它会更新模拟系统窗口中的不同统计结果（稍后将解释framesRun）：

这里指示一回合是多长时间……必须从另外某个地方发送这个参数，不过后面再做介绍。

```
private void UpdateStats(TimeSpan frameDuration) {
    Bees.Text = world.Bees.Count.ToString();
    Flowers.Text = world.Flowers.Count.ToString();
    HoneyInHive.Text = String.Format("{0:f3}", world.Hive.Honey);
    double nectar = 0;
    foreach (Flower flower in world.Flowers)
        nectar += flower.Nectar;
    NectarInFlowers.Text = String.Format("{0:f3}", nectar);
    FramesRun.Text = framesRun.ToString();
    double milliSeconds = frameDuration.TotalMilliseconds;
    if (milliSeconds != 0.0)
        FrameRate.Text = string.Format("{0:f0} ({1:f1}ms)",
                                  1000 / milliSeconds, milliSeconds);
    else
        FrameRate.Text = "N/A";
}
```

这些代码主要是从world获取数据，并更新标签。

要保证窗体上的标签名与代码中一致。

将第一个参数打印为一个无小数位的数，然后是一个空格，再打印第二个参数（有一位小数），后面是字母“ms”（并用括号括起来）。

帧速率是每秒的帧数。我们使用一个TimeSpan对象存储运行一帧需要的时间。将1000除以运行一帧所需的毫秒数，就可以得到每秒的帧数。

把这个方法增加到Form1。

创建TimeSpan对象时还会谈到这个内容。

**哇！World对象从哪来的……我们还没创建过这个对象呢，不是吗？这些时间和帧是什么意思？**

这个代码使用了十六进制转储中同样的String.Format()方法。不过，并不是使用“x2”按十六进制打印，而是使用“f3”显示一个带3位小数的数。



**下面来创建World。**

你说得对，确实需要创建World对象。向窗体的构造函数增加下面这行代码：

```
public Form1() {
  InitializeComponent();
  world = new World();
}
```

再向窗体增加一个私有World字段，名为world。

先不用管与时间有关的代码。我们已经说过，需要一种方法在World中反复运行Go()……听上去需要一种定时器。

花点时间创建一个新工程，来看看定时器是如何工作的。然后再来完成我们的模拟系统，用上你新学的知识。

# 定时器反复触发事件

还记得前面使用一个循环来模拟赛狗的动画吗？不错，还有一种更好的办法。Timer是一个非常有用的组件，可以反复触发一个事件，速率可以达到每秒一千次。

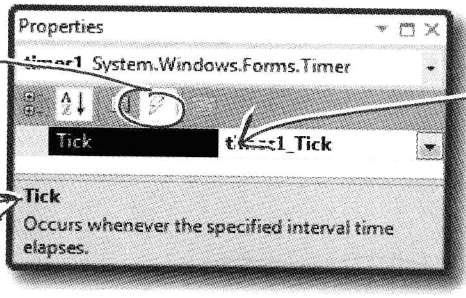
**动手做**

1. **创建一个新工程，包含一个定时器和3个按钮。**
不用关闭你当前的工程，只需打开一个新的Visual Studio，创建一个新工程。把一个定时器和3个按钮拖到窗体上。点击设计工具下面的定时器图标，设置其Interval属性为1000。这个数以毫秒为单位，这就会告诉定时器1秒触发一次tick事件。

2. **打开IDE的Properties窗口，点击Events按钮。**
(要记住，Events按钮看上去像一个闪电，可以用它管理所有窗体控件的事件）。定时器只有一个事件：Tick。点击设计工具中的Timer图标，然后在Events页中双击Tick行，IDE会为你创建一个新的事件处理方法，并自动关联到这个属性。

还可以双击Timer图标而不是使用Properties窗口来增加事件处理程序。

利用Properties窗口中的Events按钮可以处理各个控件的所有事件。




Timer控件有一个事件，名为Tick。如果双击这里，IDE会自动为你创建一个事件处理方法。

窗口下面有这个事件的一个描述。

3. **为Tick事件和按钮增加代码。**
以下代码能让你会对定时器如何工作有所认识：

```
private void timer1_Tick(object sender, EventArgs e) {
    Console.WriteLine(DateTime.Now.ToString());
}
private void toggleEnabled_Click(object sender, EventArgs e) {
    if (timer1.Enabled)
        timer1.Enabled = false;
    else
        timer1.Enabled = true;
}
private void startTimer_Click(object sender, EventArgs e){
    timer1.Start();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
private void stopTimer_Click(object sender, EventArgs e) {
    timer1.Stop();
    Console.WriteLine("Enabled = " + timer1.Enabled);
}
```

这个语句将当前日期和时间写至输出。检查输出窗口，确保每秒（每1000毫秒）触发一次tick事件。

利用这些按钮可以尝试使用Enabled属性以及Start()和Stop()方法。第一个按钮使Enabled在true和false之间切换，另外两个按钮分别调用Start()和Stop()方法。

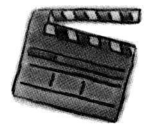
定时器的Enabled属性可以启动和停止定时器。

定时器的Start()方法启动定时器，并设置Enabled为true。Stop()方法停止定时器，并设置Enabled为false。

## 在后台定时器使用了委托

类似于处理按钮点击事件的事件处理程序，定时器的Tick事件处理程序也相当常用。

**在幕后**



C#和.NET怎样告诉定时器每次tick时做些什么？每次定时器产生tick事件时如何运行timer1 _ Tick()方法？没错，又要回到上一章讨论的事件和委托。使用IDE的“Go To Definition”特性，先来回忆事件处理程序委托是如何工作的：

**4 双击timer1变量，并选择“Go To Definition”。**

“Go To Definition”特性会让IDE自动跳到定义timer1变量的代码所在位置。IDE会跳到先前在Form1.Designer.cs中创建的相应代码，即增加timer1作为Form1对象的一个属性。向上滚动这个文件，直到找到下面这一行：

```
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

这就是定时器控件的Tick事件。设置为每1000毫秒触发一次。

System的一个委托（基本事件处理程序）。这是一个委托……它是指向一个或多个方法的指针。

这是刚才写的方法：timer1_Tick()。这里就是在告诉委托要指向这个方法。

**5 现在右键点击EventHandler，选择“Go To Definition”。**

IDE会自动跳到定义EventHandler的代码。为显示这个代码打开了一个新的标签页，请注意这个标签页的名字：“EventHandler [from metadata]”。这说明，定义EventHandler的代码不在你编写的代码中。它是.NET框架内置的，IDE会生成下面这行“假”代码，显示了这个委托如何表示：

```
public delegate void EventHandler(object sender, EventArgs e);
```

各个事件的类型为EventHandle。所以Tick事件现在指向timer1_Tick()方法。

这里解释了为什么C#中的各个事件通常都有一个Object和一个EventArgs参数，原因在于C#为事件处理定义的委托就采用了这种形式。

**BRAIN POWER**

在蜂巢模拟系统中如果每秒运行10次World的Go()方法，需要编写什么代码？

# 为模拟系统增加一个Timer

下面为模拟系统增加一个定时器。你已经有了一个定时器控件，可能命名为timer1。不过，不要使用IDE来生成timer1 _ Tick()方法，可以把这个定时器手动地绑定到一个名为RunFrame()的事件处理方法。

**DateTime &TimeSpan**

.NET使用DateTime类来存储关于时间的信息，其Now属性返回当前日期和时间。如果想得出两个时间之差，可以使用TimeSpan对象：只需将一个DateTime对象减去另一个DateTime对象，就会返回一个TimeSpan对象，其中包含两个时间之差。

TimeSpan有一些属性，如Days、Hours、Seconds和Milliseconds，可以用不同的单位度量时间。

```
public partial class Form1 : Form {
  World world;
  private Random random = new Random();
  private DateTime start = DateTime.Now;
  private DateTime end;
  private int framesRun = 0;

  public Form1() {
    InitializeComponent();
    world = new World();

    timer1.Interval = 50;
    timer1.Tick += new EventHandler(RunFrame);
    timer1.Enabled = false;
    UpdateStats(new TimeSpan());
  }

  private void UpdateStats(TimeSpan frameDuration) {
    // Code from earlier to update the statistics
  }

  public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    end = DateTime.Now;
    TimeSpan frameDuration = end - start;
    start = end;
    UpdateStats(frameDuration);
  }
}
```

根据前面应当有一个World属性。

这些用于确定某个给定时刻这个模拟系统已经运行了多长时间。

我们希望跟踪已经过去了多少帧或回合。

每50毫秒运行一次。

设置事件处理程序为我们自己的RunFrame()方法。

1秒是1000毫秒，所以定时器每秒滴答20次。

定时器关闭。

还要开始更新统计信息，指定一个新的TimeSpan（过去时间为0）。

帧数递增，并通知world调用Go()推进。

接下来，确定自上一帧以来过去的时间。

最后再次更新统计信息，并指定新的时间段。

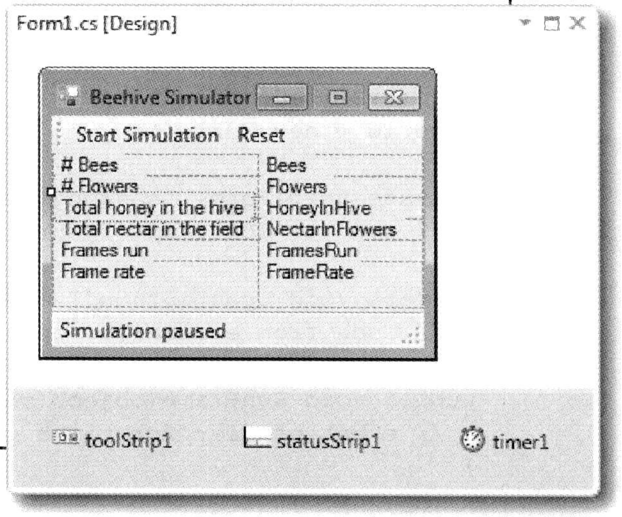
## Exercise

你的任务是为ToolStrip中的startSimulation和reset按钮编写事件处理程序。这两个按钮应当完成以下工作：

如果还没有把一个ToolStrip和StatusStrip从工具条拖到你的窗体上，现在就动手吧。

1. 开始时，第一个按钮应当标为“Start Simulation”。按下这个按钮，模拟开始，标签变为“Pause Simulation”。如果模拟暂停，按钮应当标为“Resume simulation”。

2. 第二个按钮应当标为“Reset”。按下这个按钮时，要重新创建世界。如果定时器暂停，第一个按钮的文本应当从“Resume simulation”变为“Start Simulation”。




这个问题没有唯一的答案，我们只是希望你考虑接下来要做什么。

只需在设计工具中双击ToolStrip按钮，让IDE增加它的事件处理程序，就像普通的按钮一样。

## Sharpen your pencil

已经到了模拟系统的这个阶段，你认为接下来要做什么？试着运行这个程序，在接下来介绍图形化的内容之前，你认为还要注意哪些问题，把它们写下来。

..........................................................................

..........................................................................

..........................................................................

..........................................................................

..........................................................................

..........................................................................

..........................................................................

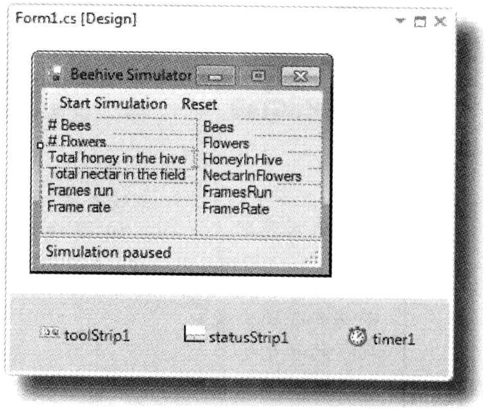
## there are no Dumb Questions

**问：我们一直在用“回合（turn）”这个词，但是你现在又提到了“帧”。它们有什么区别呢？**

**答：** 从语义上讲这两个词确实是一样的。我们还是在按回合进行处理：这是指一小段时间，在此期间世界中的每个对象都要有所动作。不过，既然马上就要讨论一些有关图形化的内容，所以我们开始使用“帧（frame）”，也就是图形化游戏中帧速率所指的“帧”。




你的任务是为Start Simulation和Reset按钮编写事件处理程序。




```
public partial class Form1 : Form {
  // variable declarations

  public Form1() {
    InitializeComponent();
    world = new World();
  }
  private void Form1_Load(object sender, tArgs e) {
    // code to start simulator
  }
  private void UpdateStats(TimeSpan frameDuration) {
    // Code from earlier to update the statistics
  }
  public void RunFrame(object sender, EventArgs e) {
    // event handler for timer
  }

  private void startSimulation_Click(object sender, EventArgs e) {
    if (timer1.Enabled) {
        toolStrip1.Items[0].Text = "Resume simulation";
        timer1.Stop();
    } else {
        toolStrip1.Items[0].Text = "Pause simulation";
        timer1.Start();
    }
  }

  private void reset_Click(object sender, EventArgs e) {
    framesRun = 0;
    world = new World();
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = "Start simulation";
  }
}
```

要保证窗体上的控件名与代码中使用的名一致。

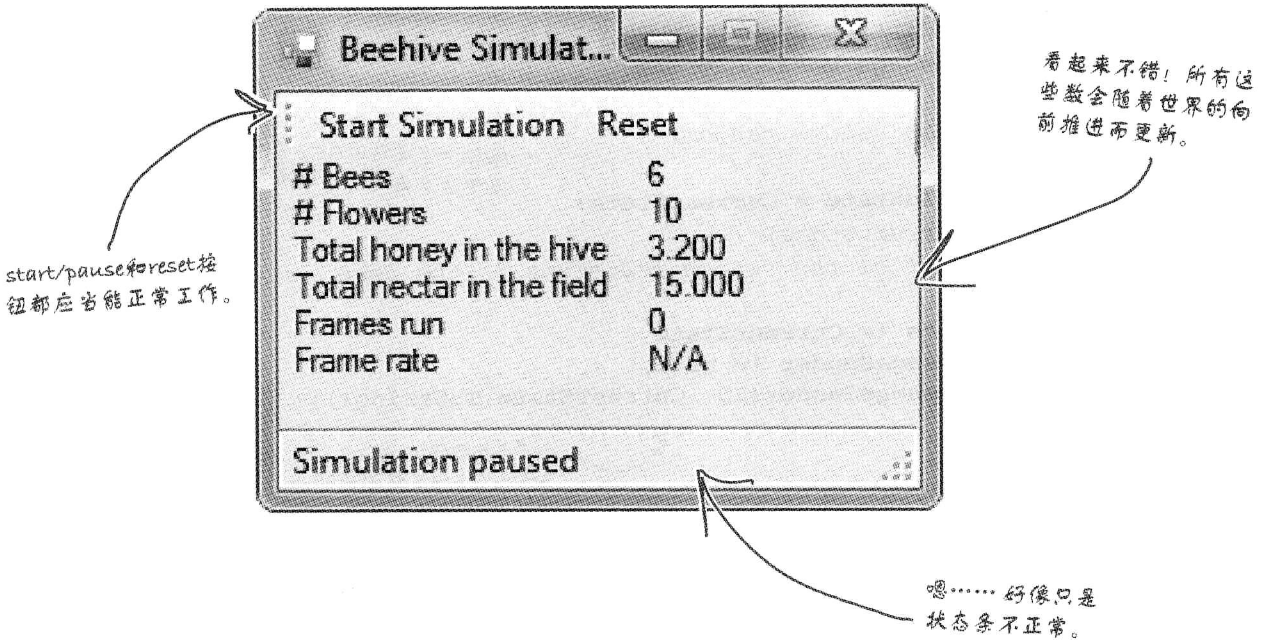
开关定时器，并更新消息。

重置模拟系统时，只需重新创建World实例，并重置framesRun。

只有当第一个按钮标签为“Resume simulation”时需要改变它的标签。如果标签为“Pause simuation”，则不必修改。

# 运行测试

前面已经做了不少工作。现在编译你的代码，修正键入错误，然后运行这个模拟系统。看上去怎么样？




**Exercise**

可以利用这个机会把你学到的东西综合起来。要让蜜蜂告诉模拟系统它们在做什么。如果蜜蜂告诉了模拟系统，我们希望模拟系统能更新状态消息。

在这里，你不仅要编写大部分代码，还要确定需要编写哪些代码。如何在每次蜜蜂改变状态时调用模拟系统中的一个方法？

给你一个小提示，我们已经编写了下面这个方法，把它增加到窗体。Bee类应当在每次改变状态时调用这个方法：

*好吧，再给你一个提示，要让它起作用，除了一个类以外，所有其他类都要有所修改。

```
private void SendMessage(int ID, string Message) {
  statusStrip1.Items[0].Text = "Bee #" + ID + ": " + Message;
}
```




你的任务是让Bee告诉模拟系统它们在做什么。

为Bee类增加的代码。

```
class Bee {
  // all our existing code
  public BeeMessage MessageSender;

  public void Go(Random random) {
    Age++;
    BeeState oldState = CurrentState;
    switch (currentState) {
     // the rest of the switch statement is the same
    }
    if (oldState != CurrentState
        && MessageSender != null)
          MessageSender(ID, CurrentState.ToString());
  }
}
```

这里使用一个回调将各个Bee对象关联到窗体的SendMessage()方法。

它使用一个名为BeeMessage的委托，这个委托取一个蜜蜂ID和一个消息作为参数。蜜蜂使用这个委托向窗体发回消息。

如果Bee的状态发生改变，就会回调BeeMessage委托指向的方法，并向该方法提供状态改变信息。

对Hive做的修改。

```
class Hive {
  // all our existing code
  public BeeMessage MessageSender;

  public Hive(World world, BeeMessage MessageSender) {
    this.MessageSender = MessageSender;
    // existing constructor code
  }

  public void AddBee(Random random) {
    // existing AddBee() code
    Bee newBee = new Bee(beeCount, startPoint, world, this);
    newBee.MessageSender += this.MessageSender;
    world.Bees.Add(newBee);
}
```

Hive也需要一个委托，从而为每个蜜蜂传递一个方法，在AddBee()中创建蜜蜂时会调用这个方法。

AddBee()现在要确保每个新蜜蜂调用了委托指向的方法。

```
public delegate void BeeMessage(int ID, string Message);
```

BeeMessage是我们建立的一个委托。它与窗体中编写的SendMessage()方法匹配。把它增加到单独的文件中（名为BeeMessage.cs），它应该在这个命名空间中，但是要在所有类以外。

World也需要做一些修改。

```
class World {
  // all our existing code

  public World(BeeMessage messageSender) {
    Bees = new List<Bee>();
    Flowers = new List<Flower>();
    Hive = new Hive(this, messageSender);
    Random random = new Random();
    for (int i = 0; i < 10; i++)
        AddFlower(random);
  }
}
```

World不需要自己的一个委托。它只是将要调用的方法传递到Hive实例。

最后一点，但不表示不重要，窗体也需要更新。这里没有显示的代码仍与以前一样。

```
public partial class Form1 : Form {
  // variable declarations

  public Form1() {
    InitializeComponent();
    world = new World(new BeeMessage(SendMessage));
    // the rest of the Form1 constructor
  }

  private void reset_Click(object sender, EventArgs e) {
    framesRun = 0;
    world = new World(new BeeMessage(SendMessage));
    if (!timer1.Enabled)
        toolStrip1.Items[0].Text = “Start simulation”;
  }

  private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = “Bee #” + ID + “: ” + Message;
  }
}
```

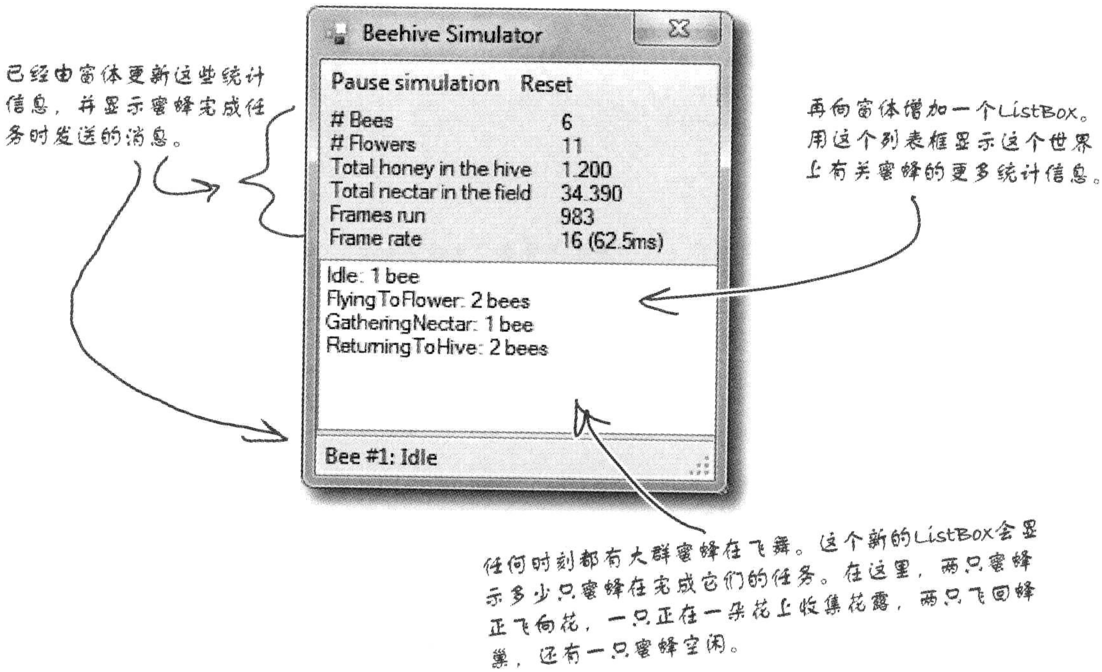
从Bee类创建一个新委托（一定要将BeeMessage声明为public），并指向我们的SendMessage()方法。

做同样的处理……创建World时，提供蜜蜂回调的方法。

这是我们在前面提供的方法……别忘了还要加入这个方法。

# 下面处理蜜蜂分组

你的蜜蜂应该已经在蜂巢和花场上嗡嗡飞舞，模拟也应该能正常运行了！是不是很酷？不过，因为这个模拟系统的可视化部分还没有完成（下一章将做这个工作），所以目前为止的所有信息都只是蜜蜂通过它们的回调发回主窗体的消息。下面来增加更多有关蜜蜂的信息，了解它们在做什么。

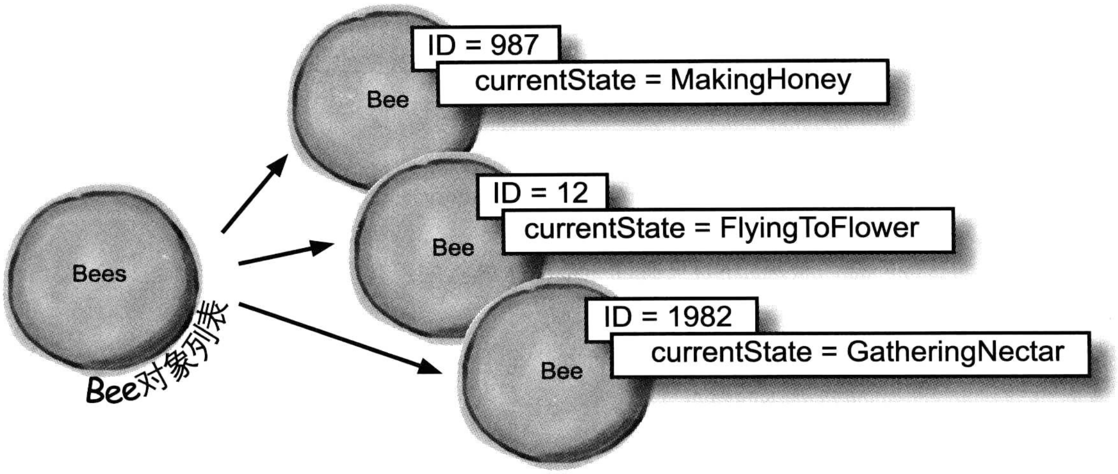



**BRAIN POWER**

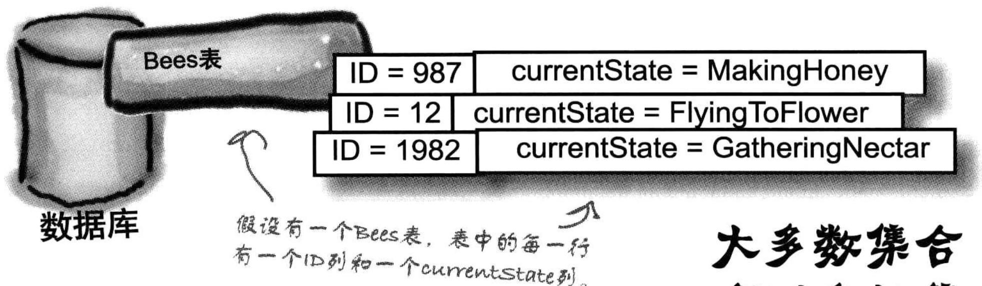
你已经很清楚如何收集信息填充ListBox，花些时间来考虑如何完成上述工作。不过，尽管看上去有些困难，但实际上并没有那么复杂。怎样才能知道处于各个不同Bee.State状态的蜜蜂有多少呢？

# 用集合来收集……数据

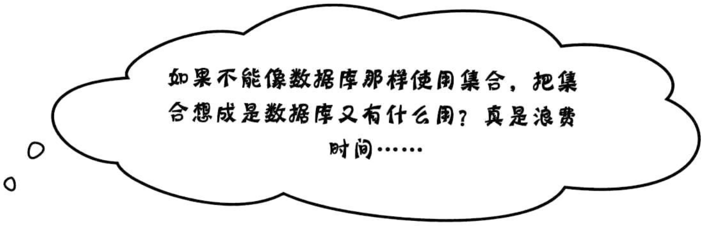
我们的蜜蜂都存储在一个List<Bee>中，这是一个集合类型。集合类型只是存储数据…… 它的工作非常类似于数据库。所以每个蜜蜂就像是一个数据行，包括状态、ID等。如下所示，蜜蜂看起来就像是一个对象集合。




Bee对象的字段中有大量数据。可以把对象集合想成是数据库中的行。每个对象的字段中包含有数据，这类似于数据库中各行的列中包含有数据。




**大多数集合（特别是包含对象的集合）都可以认为是一个数据存储库，如数据库。**






**如果能用同样的基本语法来查询集合、数据库，甚至XML文档，岂不是很妙？**

C#有一个非常有用的特性，称作LINQ ，这表示语言集成查询（Langauge INtegrated Query）。LINQ的基本思想是，对于一个数组、列表、栈、队列或其他集合，利用LINQ只通过一个操作就能处理其中的所有数据。

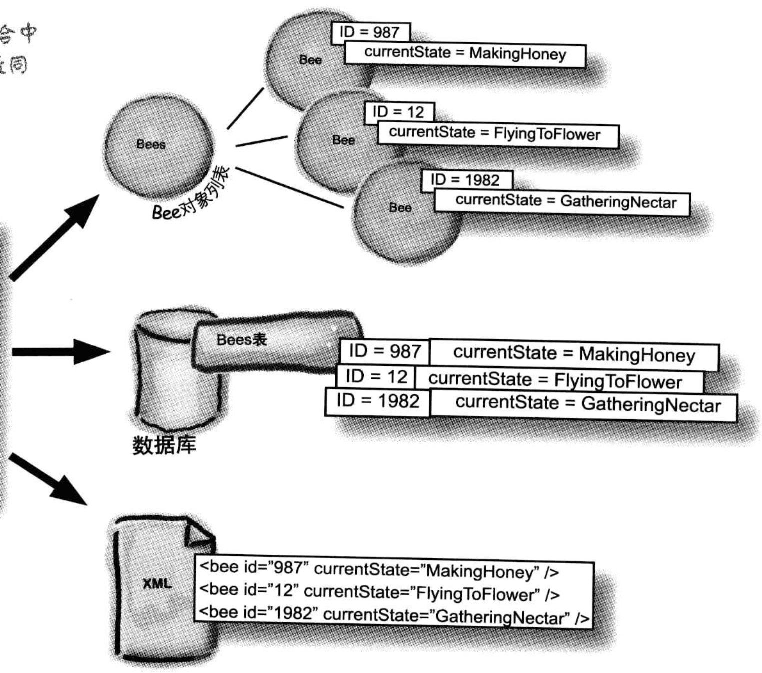
但是LINQ最突出的一点是，可以采用处理数据库的语法来处理集合。

第15章将主要介绍LINQ。

无论对数据库中的数据还是集合中的数据，这个LINQ查询都会做同样的处理。

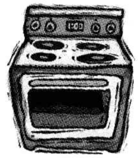
```
var beeGroups =
  from bee in world.Bees
  group bee by bee.CurrentState
  into beeGroup
  orderby beeGroup.Key
  select beeGroup;
```

LINQ



# 利用LINQ，可以很容易地处理数据库和集合中的数据

后面还会用整个一章来讨论LINQ，不过可以先使用LINQ和一些现成代码为我们的模拟系统增加另外一些特性。现成代码（Ready Bake Code）是指你可以直接键入的一些代码，即使还不完全理解也没有关系。在第15章中你将了解它是如何工作的。



**现成代码**

```
private void SendMessage(int ID, string Message) {
    statusStrip1.Items[0].Text = "Bee #" + ID + ": " + Message;
    var beeGroups =
      from bee in world.Bees
      group bee by bee.CurrentState into beeGroup
      orderby beeGroup.Key
      select beeGroup;
    listBox1.Items.Clear();
    foreach (var group in beeGroups) {
        string s;
        if (group.Count() == 1)
            s = "";
        else
            s = "s";
        listBox1.Items.Add(group.Key.ToString() + ": "
                          + group.Count() + " bee" + s);
        if (group.Key == BeeState.Idle
         && group.Count() == world.Bees.Count()
         && framesRun > 0) {
           listBox1.Items.Add("Simulation ended: all bees are idle");
           toolStrip1.Items[0].Text = "Simulation ended";
           statusStrip1.Items[0].Text = "Simulation ended";
           timer1.Enabled = false;
        }
    }
}
```

这是一个LINQ查询。它取Bees集合中的所有蜜蜂，并按其CurrentState属性分组。

这个组的键（Key）是蜜蜂的CurrentState，所以会以这个顺序在窗体上显示状态。

要确保这与窗体中列表框控件名一致。

beeGroups由LINQ查询得到。可以统计各组成员个数，然后迭代处理各个组。

这部分代码保证复数形式正确，如“1 bee”和“3 bees”。

最后，向列表框增加组状态（键）和所统计的成员个数。

这也是一个很好的特性。由于我们知道有多少个蜜蜂是空闲的……

……所以可以查看是否所有蜜蜂都是空闲的。如果是，说明蜂巢已经没有蜂蜜了，因此要停止模拟。

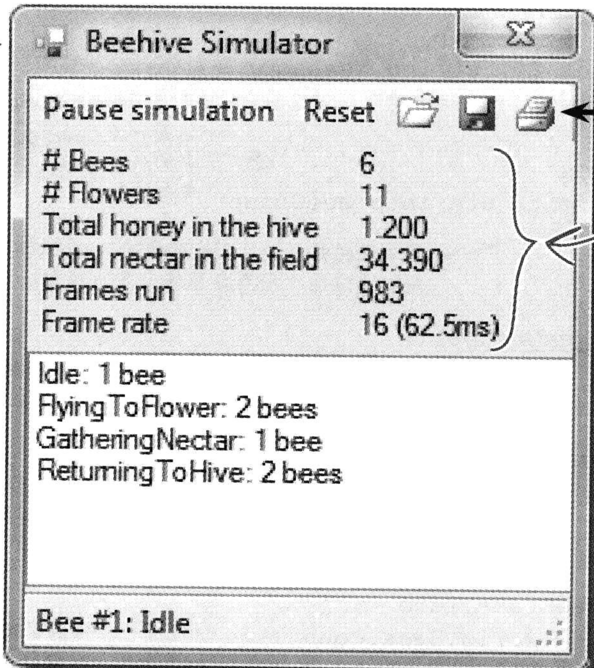



**后面几章还会对LINQ有更多了解。**

不必记住LINQ的语法，也不用现在就试图完全掌握。第15章中还会有很多机会学习如何使用LINQ。

# 运行测试(第2部分)

编译你的代码，运行这个工程。如果发现错误，请反复检查语法（特别是新增加的LINQ代码）。然后启动你的模拟系统！




窗体上的定时器控制着模拟的运行。

要增加这些标准项，以及使之正常工作的事件处理程序。

这些统计信息来自查询World对象的窗体。

每一回合LINQ都会查询你的集合来提供这些数据。

每次蜜蜂的状态改变时，蜜蜂会回调模拟系统窗体来更新窗体。

一个对象的方法关联到另一个对象中的一个委托或事件处理程序时，这将是一个引用，串行化时将跟随这个引用进一步串行化所指向的对象。

所以，如果尝试串行化一个对象，而这个对象有一个事件处理程序在监听某个控件上的一个事件，倘若没有置为[NonSerialized]，它就会串行化这个控件，这会抛出一个SerializationException。

## [NonSerialized] 避免数据串行化

有时你希望串行化对象的一部分，而不是全部。对象中可能包含你并不希望写至磁盘的数据。假设你在构建一个要求用户登录的系统，希望把存储用户选项和设置的对象保存到一个文件中。可以用[NonSerialized]属性来标志password字段。这样一来，使用Serialize()串行化这个对象时，就会跳过这个字段。

如果对象有一个引用指向另一个不可串行化的对象，[NonSerialized]会特别有用。例如，如果试图串行化一个Form，Seralize()会抛出一个SerializationException异常。所以如果对象有一个指向Form对象的引用，试图完成串行化时，串行化方法会沿着这个链接尝试串行化Form……这就会抛出SerializationException异常。但是如果用[NonSerialized]属性标志包含这个引用的字段，Serialize()根本不会跟随这个引用去串行化Form对象。

# 最后一个难题:打开和保存

我们的工作基本上已经完成，接下来可以转向图形化部分，为这个模拟系统增加一些抢眼的内容。不过，还要为这个版本再增加一点特性：允许加载、保存和打印蜜蜂的统计信息。

**1** **增加Open、Save和Print图标。**
ToolStrip控件有一个很有用的特性，它能自动地为标准图标插入图片按钮，包括new、open、save、print、cut、copy、paste和help。只需右键点击窗体设计窗口下面的ToolStrip图标，并选择“Insert Standard Items”。然后点击第一项（也就是“new”图标），将其删除。保留后面3项，因为这是我们需要的(open、save和print)，然后是一个分隔线；可以删除这个分隔线，或者把它移到Reset按钮和save按钮之间。然后删除其余的按钮。一定要把它的CanOverflow属性设置为false（这样就不会在工具条右侧增加一个溢出菜单按钮），另外GripStyle属性要设置为Hidden（从而从左侧删除大小调整框）。

现在要增加Print按钮，下一章将通过这个按钮为蜂巢打印一个状态页。

**2** **增加按钮事件处理程序。**
这些新的标准按钮名为openToolStripButton、saveToolStripButton和 printToolStripButton。只需双击这些按钮增加它们的事件处理程序。



**Exercise**

增加代码使save和open按钮能正常工作。

1. **让save按钮将world串行化到一个文件。** save按钮要停止定时器（可以在保存后再重启）。它要显示一个Save对话框，如果用户指定了一个文件名，则要串行化World对象以及已经运行的帧数。

串行化World对象时，它会抛出一个SerializationException异常，并给出以下消息：`Type 'Form1'is not marked as serializable`(类型、Form1、未标志为可串行化）。这是因为串行化方法发现BeeMessage的一个字段就会沿着这个字段寻找要串行化的对象。由于这个委托关联到窗体上的一个字段，所以串行化方法就会试图串行化窗体。

要修正这个问题，只需向Hive和Bee类中的**MessageSender**字段增加**[NonSerialized]**属性，这样.NET就不会试图串行化委托指向的代码了。

2. **让open按钮从一个文件逆串行化world。** 对定时器的处理与在save按钮中类似，弹出一个Open对话框，从所选的文件逆串行化world和已经运行的帧数。然后可以再次关联MessageSender委托，并重启定时器（如果必要）。

3. **不要忘记异常处理！** 如果读写文件有问题，world应当不做任何改变。可以弹出让人可读的一个错误消息，指出哪里出了问题。

**Exercise Solution**

你的任务是让Save和Open按钮能正常工作。

不要忘记增加这些using语句。

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

要保证World, Hive, Flower和Bee类可以串行化。串行化world时，.NET会查找它的Hive, Flower和Bee对象引用，并完成这些对象的串行化。

```
[Serializable]
class World {

[Serializable]
class Hive {

[Serializable]
class Flower {

[Serializable]
class Bee {
```

确保Hive和Bee类中的MessageSender字段标志有[NonSerialized]。

```
[NonSerialized]
public BeeMessage MessageSender;
```

这是Save按钮的代码。

```
private void saveToolStripButton_Click(object sender, EventArgs e) {
    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();

    SaveFileDialog saveDialog = new SaveFileDialog();
    saveDialog.Filter = "Simulator File (*.bees)|*.bees";
    saveDialog.CheckPathExists = true;
    saveDialog.Title = "Choose a file to save the current simulation";
    if (saveDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream output = File.OpenWrite(saveDialog.FileName)) {
                bf.Serialize(output, world);
                bf.Serialize(output, framesRun);
            }
        }
        catch (Exception ex) {
            MessageBox.Show("Unable to save the simulator file\r\n" + ex.Message,
              "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    if (enabled)
        timer1.Start();
}
```

这里使用“.bees”作为模拟系统保存文件的扩展名。

这里将world写入一个文件。

要记住，串行化World时，它引用的所有对象也会串行化……包括所有蜜蜂、花和蜂巢。

保存文件后，可以重启定时器（如果之前停止了定时器）。

这是Open按钮的代码。

```
private void openToolStripButton_Click(object sender, EventArgs e) {
    World currentWorld = world;
    int currentFramesRun = framesRun;

    bool enabled = timer1.Enabled;
    if (enabled)
        timer1.Stop();

    OpenFileDialog openDialog = new OpenFileDialog();
    openDialog.Filter = "Simulator File (*.bees)|*.bees";
    openDialog.CheckPathExists = true;
    openDialog.CheckFileExists = true;
    openDialog.Title = "Choose a file with a simulation to load";
    if (openDialog.ShowDialog() == DialogResult.OK) {
        try {
            BinaryFormatter bf = new BinaryFormatter();
            using (Stream input = File.OpenRead(openDialog.FileName)) {
                world = (World)bf.Deserialize(input);
                framesRun = (int)bf.Deserialize(input);
            }
        }
        catch (Exception ex) {
    MessageBox.Show("Unable to read the simulator file\r\n" + ex.Message,
    "Bee Simulator Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            world = currentWorld;
            framesRun = currentFramesRun;
        }
    }

    world.Hive.MessageSender = new BeeMessage(SendMessage);
    foreach (Bee bee in world.Bees)
        bee.MessageSender = new BeeMessage(SendMessage);
    if (enabled)
        timer1.Start();
}
```

打开文件并读文件之前，保存当前world和framesRun的一个引用。如果有问题，可以恢复为当前的世界和运行帧数并继续运行。

创建"打开文件"对话框，并弹出这个对话框。

using确保流会关闭。

在这里从文件逆串行化world和已运行的帧数。

如果文件操作抛出一个异常，则恢复当前world和framesRun。

一旦完成加载，再次关联委托，重启定时器。

**学习下一章之前，你的模拟系统应该已经能启动并运行。可以从 Head First Labs网站（www.headfirstlabs.com/books/hfcsharp/）下载一个可以运行的版本。**

# 13 控件与图片

# 应用的美化



**有时需要你自己控制图形化。**

大多数情况下，我们都依赖于控件处理应用的可视化部分。不过有时这还不够，比如说，如果想对一个图片完成动画，就需要自己来控制。一旦开始实现动画，就要为.NET程序创建你自己的控件，可能需要增加双缓冲，甚至直接在窗体上绘图。一切都要从Graphics对象和Bitmap开始，还要有决心摒弃既有的图形化处理。

# 你一直在使用控件与程序交互

TextBox、PictureBox、Label……你现在应该已经很清楚该如何使用IDE工具条中的控件。不过，你对它们真的了解吗？对于控件，所能做的绝不仅仅是把一个图标拖到窗体中。

**可以创建你自己的控件。**

工具条中的控件对于构建窗体和应用确实非常有用，不过它们并没有什么神奇。这些控件也只是一些类，就像你前面编写的类一样。实际上，利用C#，创建你自己的控件相当容易，只需继承适当的基类。

**你的定制控件可以出现在IDE的工具条上。**

IDE的工具条也没有什么神奇之处。它只是查看工程中的类以及内置的.NET类来查找控件。如果找到了一个控件（也就是实现了适当接口的一个类），就会显示该类的一个图标。如果你增加了自己的定制控件，它们也会出现在工具条上。

可以继承现有的任何一个控件类来创建一个新类，即使它没有增加任何其他代码，也会自动显示在工具条上。

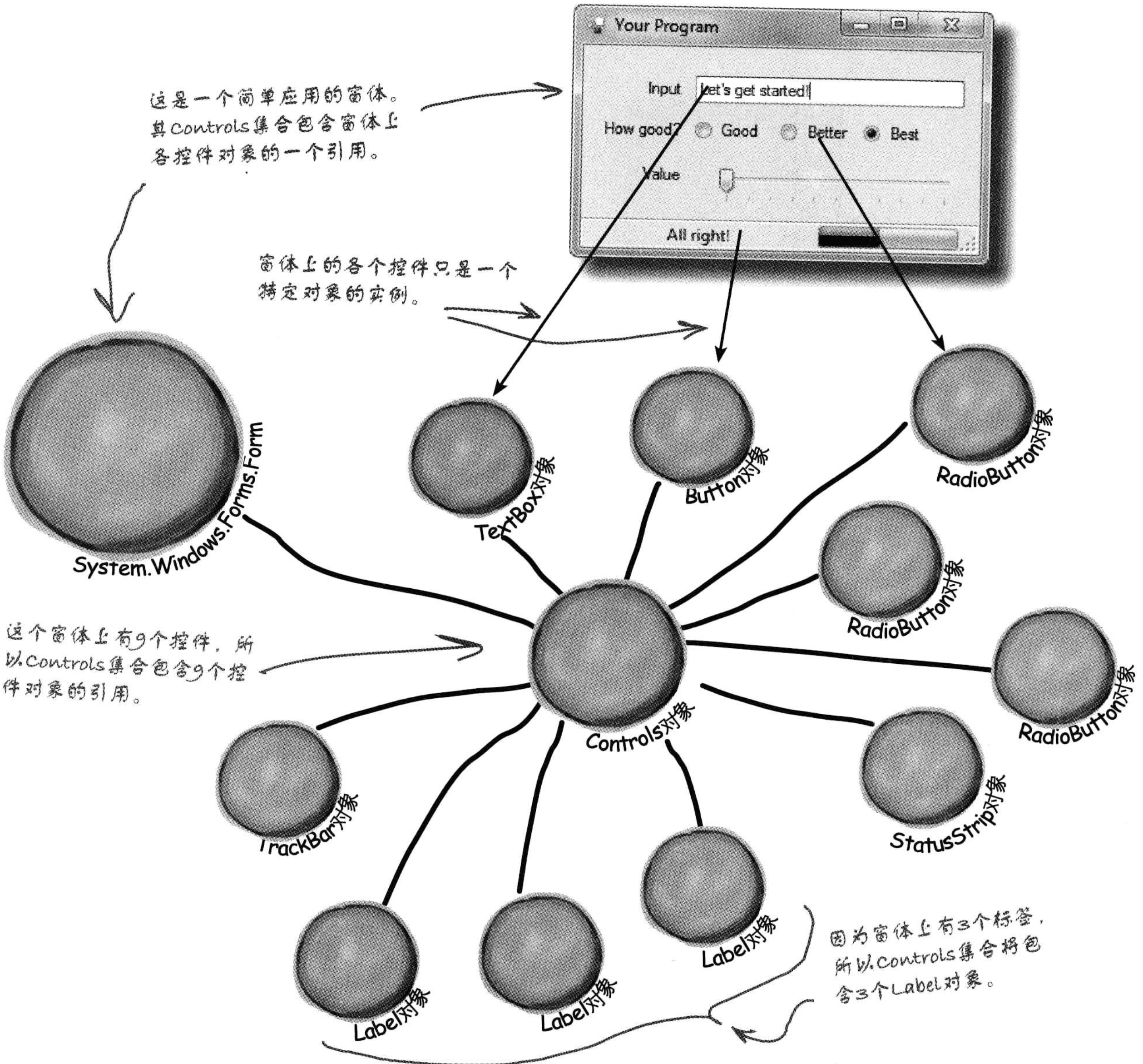
**可以编写代码在程序运行时向窗体增加控件，甚至删除控件。**

可以在IDE窗体设计工具中显示窗体，但这并不表示窗体一成不变。你已经实现过PictureBox控件的移动（如构建赛狗应用时，就是通过移动图片来表示狗的奔跑）。不过，除此以外还可以增加或删除控件。实际上，在IDE中构建一个窗体时，所做的正是编写代码向窗体增加控件…… 这说明可以编写类似的代码，并在需要时运行这些代码。

# 窗体控件也只是对象

你已经知道了控件对于窗体何等重要。从第1章开始，就一直在使用按钮、文本框、图片框、复选框、分组框、标签和其他窗体。可以看到，这些控件都只是一些对象，与你处理的所有其他对象是一样的。

控件就是对象，与其他对象类似，只不过它知道如何自行绘制。Form对象使用一个特殊的集合Controls来记录它的控件，可以利用这个集合在你自己的代码中增加或删除控件。

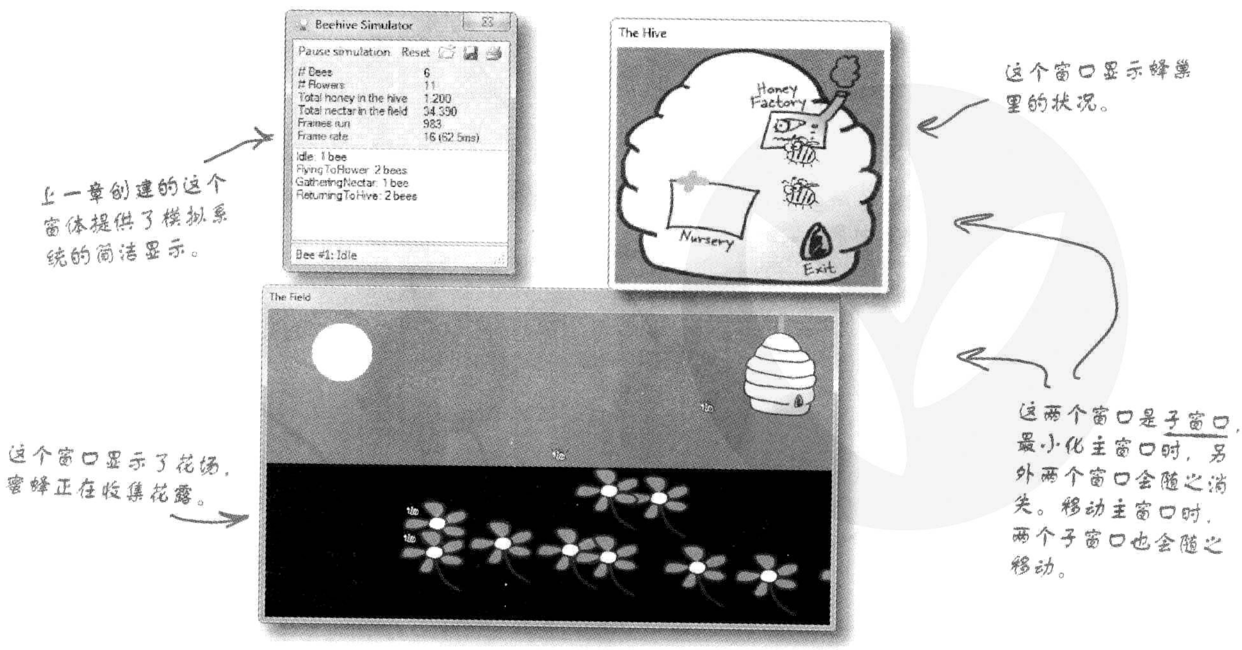


# 使用控件完成蜂巢模拟系统的动画

你已经构建了一个很不错的模拟系统，但是看起来还不太漂亮。下面来创建一个真正炫的可视化应用，它能显示蜜蜂的实际动作。你要构建一个表现工具来完成蜂巢的动画……这里的关键就是控件。

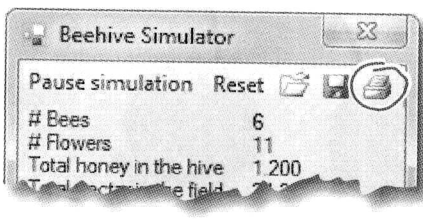
**1 用户界面显示所发生的当前状况。**

模拟系统将有3个不同的窗口。你已经建立了其中的主窗口，即一个“简洁”的统计信息窗口，显示当前的有关统计信息，并根据蜜蜂的状态实时更新。现在要增加一个窗口来显示蜂巢里的状况，另外增加一个窗口显示蜜蜂收集花露的花场。



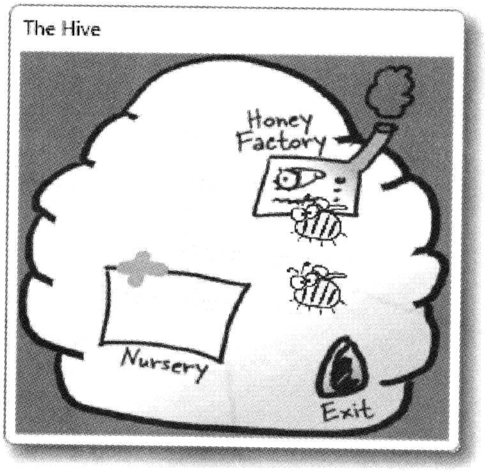

**2 向统计信息窗口增加一个Print按钮。**

统计信息窗口已经有一个Open按钮和一个Save按钮，Print按钮还不起作用。可以重用前面的大量图形代码让ToolStrip上的Print按钮打印一个信息页，显示当前的状况。



**3** **蜂巢窗口显示蜂巢里的状况。**

蜜蜂在这个世界里飞来飞去，需要完成各个蜜蜂的动画。有时它们已经在蜂巢里，如果蜜蜂在蜂巢里，就要在这个窗口中显示。



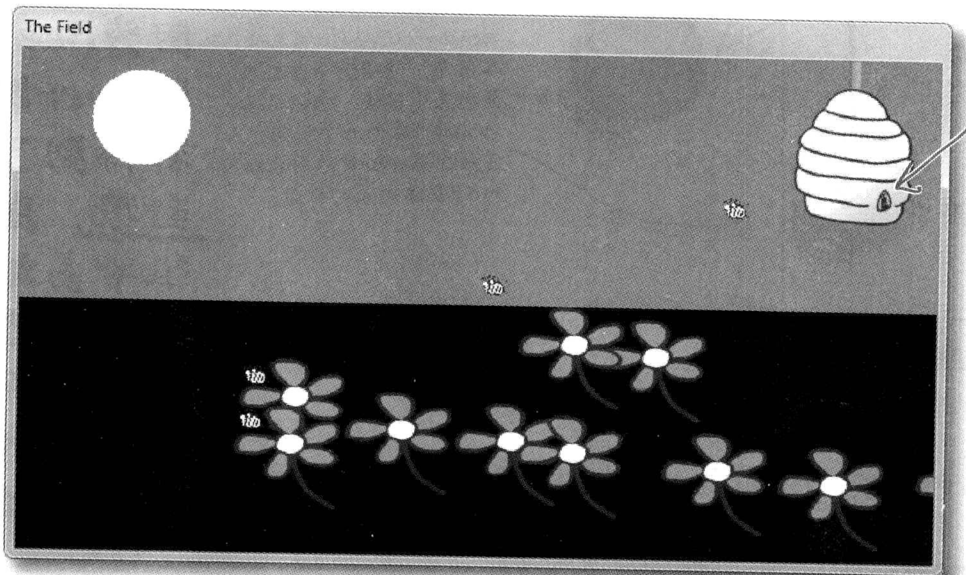

蜂巢里有3个重要位置。蜜蜂在保育室出生，它们必须飞到出口才能离开蜂巢从花收集花露，回到蜂巢后需要到蜂蜜工厂将花露酿成蜂蜜。

蜂巢出口在蜂巢窗体上，入口在花场窗体上（所以Hive的位置字典中既有入口也有出口）。

**4** **蜜蜂在花场窗口中收集花露。**

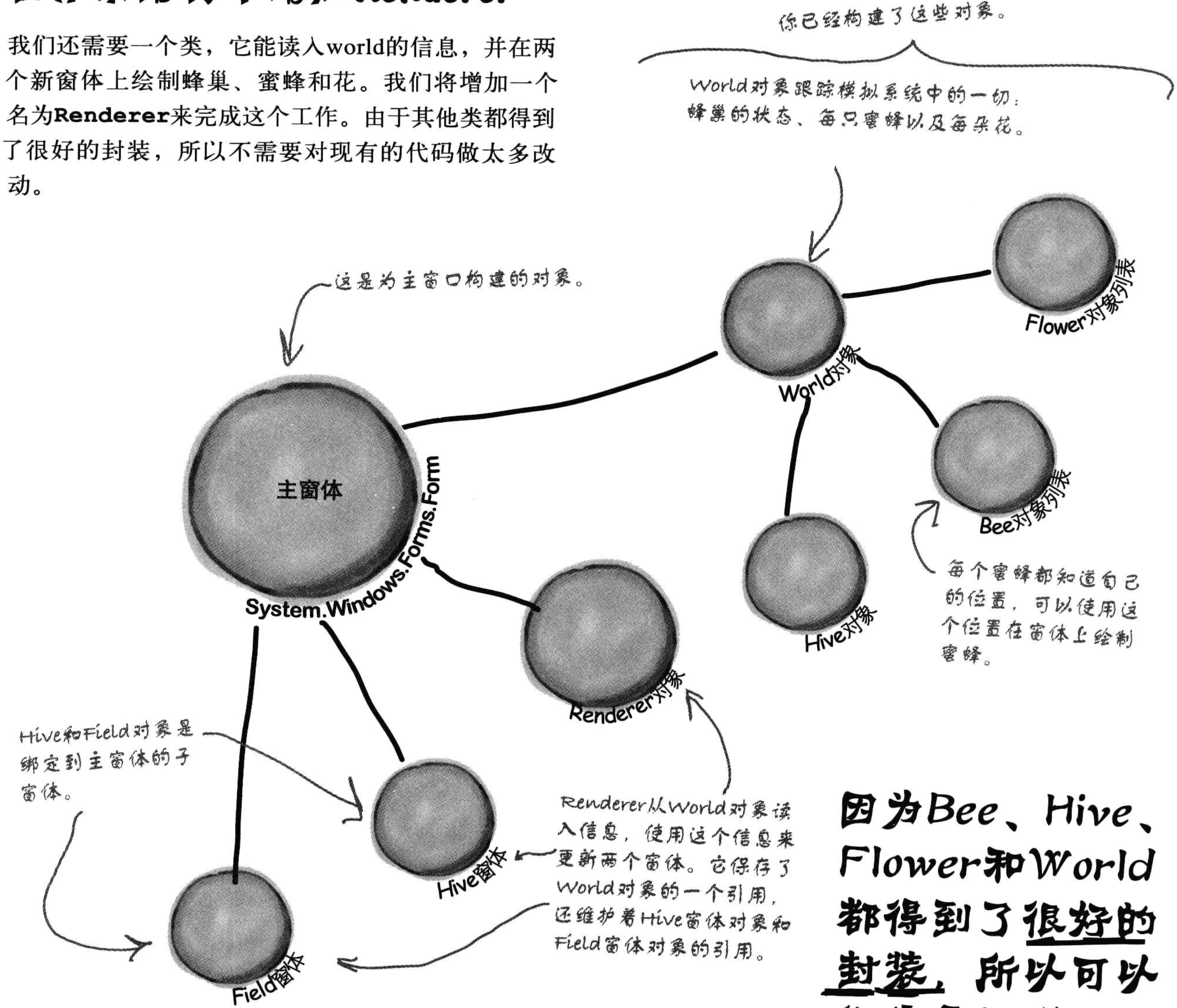
蜜蜂有一个重要的任务：就是从花收集花露，并带回蜂巢酿成蜂蜜。然后吃蜜来提供能量，再飞出蜂巢收集更多花露。

这是回蜂巢的入口。蜜蜂飞入这里时，会从花场窗体消失，再次出现在蜂巢窗体中靠近出口的位置。



# 在体系结构中增加Renderer

我们还需要一个类，它能读入world的信息，并在两个新窗体上绘制蜂巢、蜜蜂和花。我们将增加一个名为**Renderer**来完成这个工作。由于其他类都得到了很好的封装，所以不需要对现有的代码做太多改动。

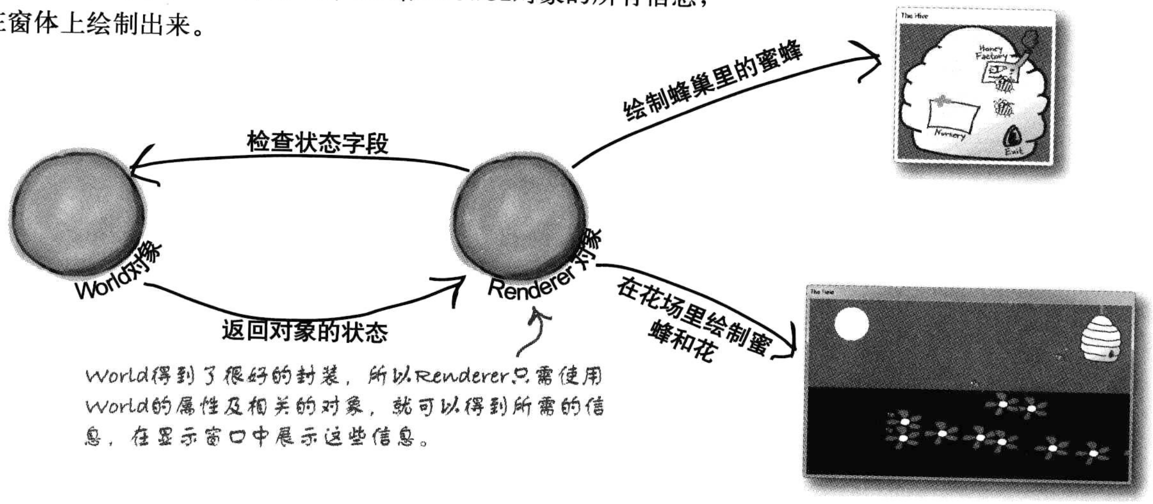



**因为Bee、Hive、Flower和World都得到了很好的封装，所以可以很容易地增加一个类来表现这些对象，而不需要对现有的代码做太多改动。**

> **ren-der, 动词。**
> 以艺术手段表示或表现。Sally的艺术老师要求全班观察模特的阴影和线条，并在纸上表现出来。

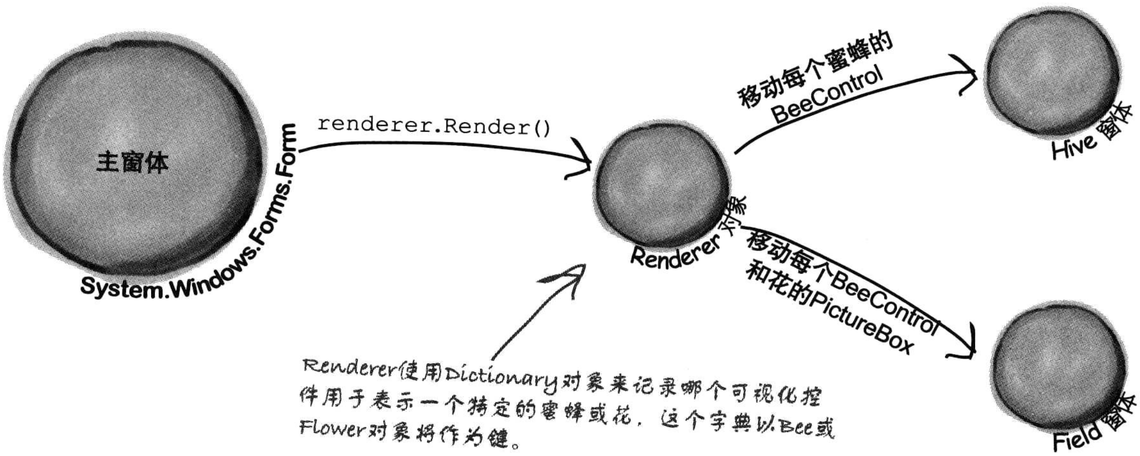
# Renderer在两个窗体上绘制World中的一切

World对象记录了模拟系统中的一切，包括蜂巢、所有蜜蜂和花。但是它并没有具体绘图，也没有生成任何输出。这是Renderer对象的任务。这个对象读入World、Hive、Bee和Flower对象的所有信息，并在窗体上绘制出来。




## 模拟系统每运行一帧后都会展示这个世界

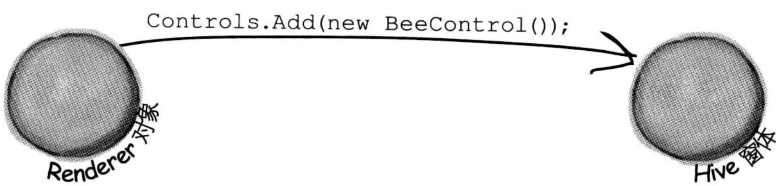
主窗体调用world的Go()方法后，应当调用renderer的Render()方法重新绘制显示窗口。例如，每朵花用一个PictureBox控件显示。不过下面对蜜蜂做进一步处理，将创建一个动画控件。你要创建这个新控件（名为BeeControl），并自己定义它的行为。



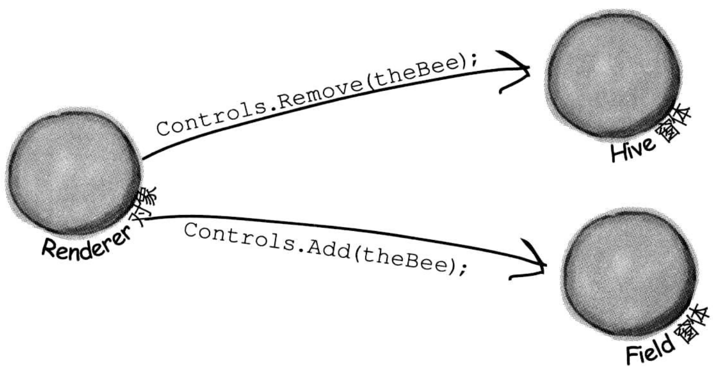
# 控件很适合表示可视化显示元素

向蜂巢增加一个新的蜜蜂时，我们希望模拟系统向Hive窗体增加一个新的BeeControl，另外它在这个世界中移动时能改变它的位置。蜜蜂飞出蜂巢时，模拟系统需要从Hive窗体删除这个控件，把它增加到Field窗体。当它带着花露飞回蜂巢时，则需要从Field窗体删除这个控件，再增加到Hive窗体。而且我们希望这个动画蜜蜂图片能一直扇动翅膀。利用控件很容易达到以上的要求。

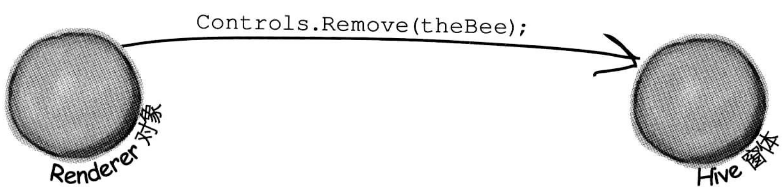
❶ world增加一个新蜜蜂，renderer创建一个新的BeeControl，并把它增加到Hive窗体的Controls集合。




❷ 蜜蜂飞出蜂巢进入花场时，renderer将这个BeeControl从蜂巢的Controls集合删除，增加到Field窗体的Controls集合。




❸ 蜜蜂空闲而且太老时，就会退休。renderer会检查world的Bees列表，如果发现这个蜜蜂已经删除，就会从Hive窗体删除相应控件。



## Sharpen your pencil

你能看出这些代码都有什么作用吗？假设所有代码段都放在一个窗体中，尽可能猜出正确的答案。

```
this.Controls.Add(new Button());
```

```
Form2 childWindow = new Form2();
childWindow.BackgroundImage =
          Properties.Resources.Mosaic;
childWindow.BackgroundImageLayout =
          ImageLayout.Tile;
childWindow.Show();
```

如果窗体中有一个ListBox，可以使用其AddRange()方法增加列表项。

```
Label myLabel = new Label();
myLabel.Text = "What animal do you like?";
myLabel.Location = new Point(10, 10);
ListBox myList = new ListBox();
myList.Items.AddRange( new object[]
       { "Cat", "Dog", "Fish", "None" } );
myList.Location = new Point(10, 40);
Controls.Add(myLabel);
Controls.Add(myList);
```

不需要写出每一行，只要能总结出代码块做了什么就可以了。

```
Label controlToRemove = null;
foreach (Control control in Controls) {
    if (control is Label
           && control.Text == "Bobby")
         controlToRemove = control as Label;
}
Controls.Remove(controlToRemove);
controlToRemove.Dispose();
```

**附加题：你认为为什么没有把Controls.Remove()语句放在foreach循环里？**

如果愿意，可以自己试试看，.NET会提供一个结果，你认为为什么会有这样的结果，请写出原因。

## Sharpen your pencil Solution

你能看出这些代码都有什么作用吗？假设所有代码段都放在一个窗体中，尽可能猜出正确的答案。

```
this.Controls.Add(new Button());
```

创建一个新按钮，并增加到窗体。它使用默认值（例如，Text属性将为空）。

```
Form2 childWindow = new Form2();
childWindow.BackgroundImage =
          Properties.Resources.Mosaic;
childWindow.BackgroundImageLayout =
          ImageLayout.Tile;
childWindow.Show();
```

这是应用中第二个Form，名为Form2，所以这个代码会创建这个窗体，设置其背景图像为一个名为"Mosaic"的资源图像，让图像平铺而不是拉伸，然后向用户显示这个窗口。

```
Label myLabel = new Label();
myLabel.Text = "What animal do you like?";
myLabel.Location = new Point(10, 10);
ListBox myList = new ListBox();
myList.Items.AddRange( new object[]
       { "Cat", "Dog", "Fish", "None" } );
myList.Location = new Point(10, 40);
Controls.Add(myLabel);
Controls.Add(myList);
```

这个代码创建一个新标签，设置其文本，把它移到一个新位置。然后创建一个新的列表框，在其中增加4个列表项，将这个列表框移到标签的下面。将标签和列表框增加到窗体，并立即显示。

如果Controls集合中没有名为"Bobby"的控件会发生什么情况？

```
Label controlToRemove = null;
foreach (Control control in Controls) {
    if (control is Label
          && control.Text == "Bobby")
        controlToRemove = control as Label;
}
Controls.Remove(controlToRemove);
controlToRemove.Dispose();
```

这个循环搜索窗体上的所有控件，直到找到一个文本为"Bobby"的标签。一旦发现这个标签，把它从窗体上删除。

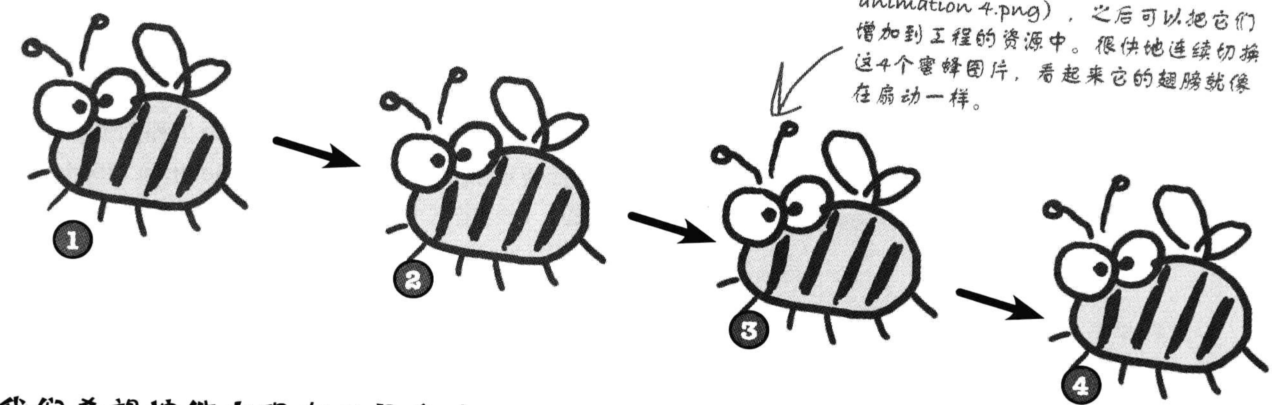
如果把这个语句放在foreach循环中，.NET会抛出一个异常。它要求集合不能改变，否则无法控制，会得到不可预料的结果。所以这里要使用for循环。

**附加题：你认为为什么不能把Controls.Remove()语句放在foreach循环中？**

在迭代处理Controls集合（或任何其他集合）的foreach循环中间不能改变这个集合。

# 构建第一个动画控件

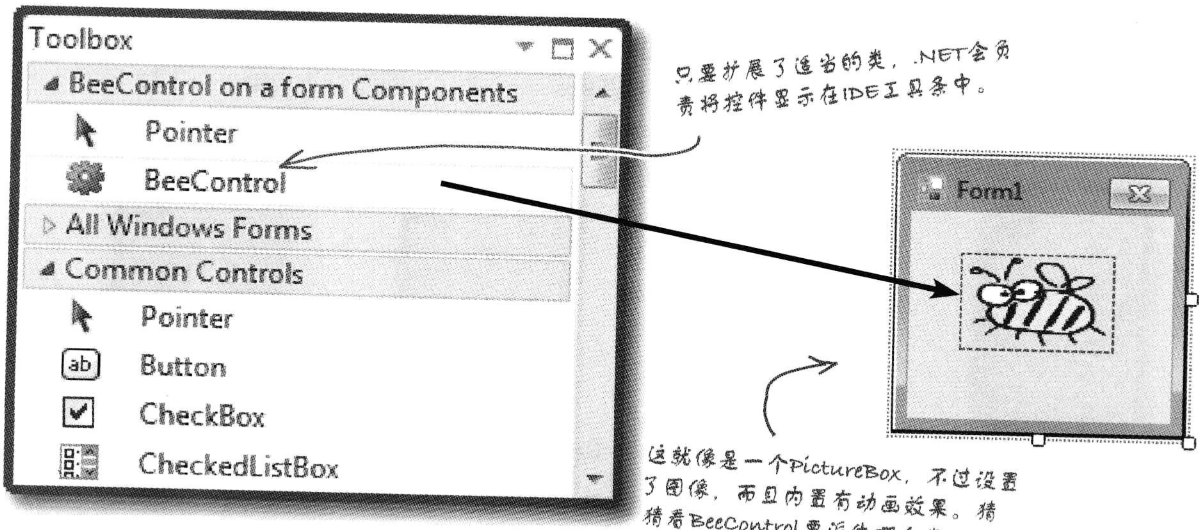
下面来构建你自己的控件，绘制一个动画的蜜蜂图片。即使你从来没有做过动画，也不用担心，它并没有听上去那么难：动画就是一个接一个地绘制一组图片，从而产生一种运动的假象。幸运的是，基于C#和.NET处理资源的方式，实现动画相当容易。




## 我们希望控件出现在工具条中

如果正确地构建了`BeeControl`，它会作为一个控件出现在工具条中，可以从工具条拖出放在窗体上。看上去就是一个显示蜜蜂图片的`PictureBox`，只不过它有动画效果，翅膀能够扇动。

**从Head First Labs网站下载这一章的图片：www.headfirstlabs.com/books/hfcsharp/**



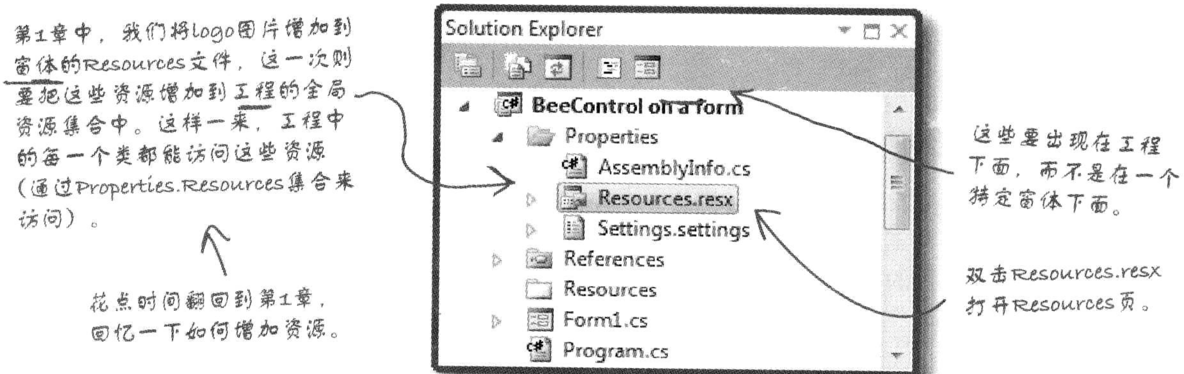
# BeeControl像一个PictureBox…… 所以首先继承PictureBox

由于工具条中的每个控件都只是一个对象，所以建立一个新控件很容易。只需向工程增加一个新类，继承一个现有的控件，并增加希望这个控件完成的所有新行为。

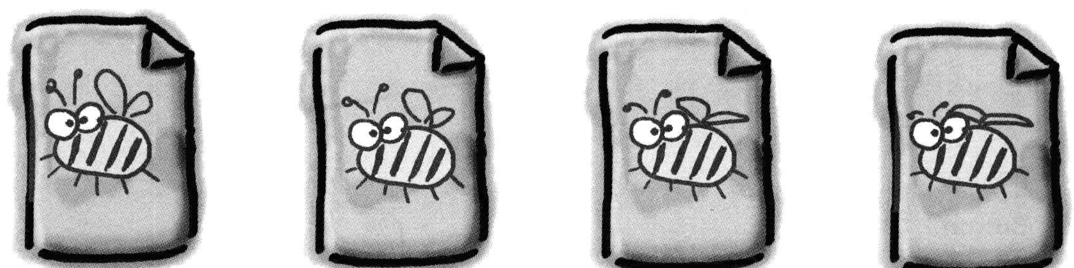
我们希望一个控件（假设名为BeeControl）显示动画的蜜蜂图片，一个蜜蜂在扇动它的翅膀，但是首先要从显示非动画图片的控件起步，然后再增加动画。所以我们先从PictureBox开始，再增加一些代码，在这个控件上画一个动画的蜜蜂。

做动画！

1. **创建一个新工程**，类似于第1章中向工程增加Objectville纸业公司的logo，为这个工程的资源增加4个动画元素。不过，这里并非增加到窗体资源，而要增加到工程的资源中。在Solution Explorer中找到工程的Resources.resx文件（在Properties下面）。双击这个文件，打开工程的Resources页。




2. 我们已经绘制了4个蜜蜂动画图片，这些图片可以从http://www.headfirstlabs.com/books/hfcsharp/下载，导入到你的资源中。然后打开Resources页，从屏幕最上面第一个下拉列表中选择“Images”，再从“Add Resource”下拉列表选择“Add Existing File…”。



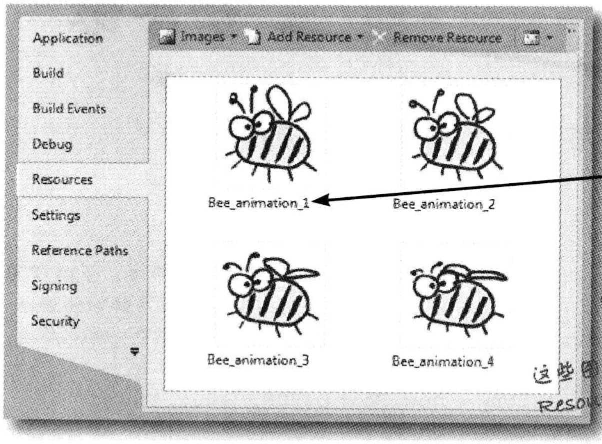
Bee animation 1.png　Bee animation 2.png　Bee animation 3.png　Bee animation 4.png

将这些图片导入到工程的资源中。

❸ 向工程的资源文件增加图片或其他资源时，可以使用Properties.Resources类访问。只需在代码中的任意一行上键入Properties.Resources——键入这个代码后，智能提示窗口就会弹出一个下拉列表，显示导入的所有图片。

程序运行时，各图片作为Bitmap对象存储在内存中。

注意最后的“.”……正是这个点告诉IDE弹出智能提示窗口，列出这个类的属性和方法。




```
pictureBox1.Image =
    Properties.Resources.Bee_animation_1;
```

这里设置一个特定PictureBox使用的图片（这也是我们的起始图片）。

这些图片存储为Properties.Resources类的公共属性。

需要为PictureBox和Timer增加一行“using System.Windows.Forms”。

❹ **现在增加你的BeeControl！** 只需把这个BeeControl类增加到工程中：

```
class BeeControl : PictureBox {
  private Timer animationTimer = new Timer();
  public BeeControl() {
    animationTimer.Tick += new EventHandler(animationTimer_Tick);
    animationTimer.Interval = 150;
    animationTimer.Start();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
  }

  private int cell = 0;
  void animationTimer_Tick(object sender, EventArgs e) {
    cell++;
    switch (cell) {
      case 1: BackgroundImage = Properties.Resources.Bee_animation_1; break;
      case 2: BackgroundImage = Properties.Resources.Bee_animation_2; break;
      case 3: BackgroundImage = Properties.Resources.Bee_animation_3; break;
      case 4: BackgroundImage = Properties.Resources.Bee_animation_4; break;
      case 5: BackgroundImage = Properties.Resources.Bee_animation_3; break;
      default: BackgroundImage = Properties.Resources.Bee_animation_2;
      cell = 0; break;
    }
  }
}
```

一定要在类文件最前面增加“using System.Windows.Forms”。

在这里初始化定时器，首先实例化定时器，并设置Interval属性，然后增加它的tick事件处理程序。

一旦回到第1帧，要把cell重置为0。

每次定时器的tick事件触发时，会让cell增加，然后根据cell执行一个switch语句，将适当的图片赋至Image属性（这个属性是从PictureBox继承的）。

修改一个控件的代码时，需要重新构建程序，这样所做的改动才会出现在设计工具中。

**然后重新构建程序。** 回到窗体设计工具，查看工具条，现在已经有BeeControl了。把它拖到你的窗体上，这样就得到了一个动画的蜜蜂！

# 创建一个按钮向窗体增加BeeControl

向窗体增加控件很简单,只需把它增加到Controls集合。从窗体中删除控件也很简单,从Controls集合删除就可以了。不过控件实现了IDisposable，所以要确保删除控件后一定要将其撤销。

**动手做**

1. **从窗体删除BeeControl，然后增加一个按钮。**
   在窗体设计工具中从窗体删除**BeeControl**。然后增加一个按钮，我们将让这个按钮来增加和删除一个BeeControl。

2. **增加一个按钮来增加和删除蜜蜂控件。**
   这个控件的事件处理程序如下:

向Controls集合增加一个控件时，它会立即出现在窗体上。

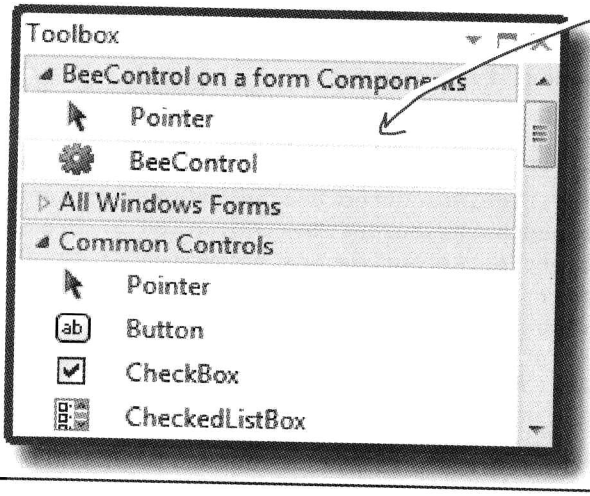
实例化后，可以使用一个对象初始化方法设置BeeControl属性。

```
BeeControl control = null;
private void button1_Click(object sender, EventArgs e) {
    if (control == null) {
        control = new BeeControl() { Location = new Point(100, 100) };
        Controls.Add(control);
    } else {
        using (control) {
          Controls.Remove(control);
        }
        control = null;
    }
}
```

我们利用一个using语句来确保从Controls集合删除控件后会将其撤销。

现在运行程序时，点击一次按钮时，它会将一个新的BeeControl增加到窗体上。再次点击这个按钮，将把这个控件删除。这里使用私有control字段保存这个控件的引用（窗体上没有控件时则设置这个引用为null）。

**在幕后**




只需创建继承Control的一个类，就可以向工具条增加你自己的控件。

**工具条中的每个可视化控件都继承自System.Windows.Forms.Control。你现在对于这个类实现的一些成员应该已经不陌生了：Visible，Width，Height，Text，Location，BackColor，BackgroundImage……在任何控件的Properties窗口中都能看到所有这些熟悉的属性。**

# 控件还需要撤销它们的控件！

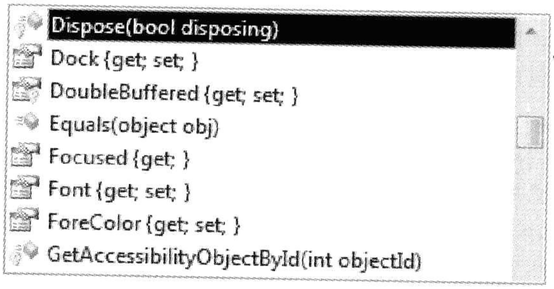
BeeControl还存在一个问题。控件删除后需要撤销。但是BeeControl创建了Timer的一个新实例，这也是一个出现在工具条中的控件……它并没有被撤销！这就有问题了。幸运的是，这个问题很容易修正，只需覆盖Dispose()方法。

控件类实现了IDisposable，所以要保证所用的每一个控件都得到撤销。

**3 覆盖Dispose()方法，撤销定时器。**

由于BeeControl继承自一个控件，这个控件必然有一个Dispose()方法。所以只需覆盖并扩展这个方法来撤销定时器。找到控件的代码，键入override：

```
class BeeControl : PictureBox {
    override
```




在一个类中键入"override"时，IDE会弹出一个智能提示窗口，其中列出可以覆盖的所有方法。选择Dispose()方法，将自动为你填写！

只要点击Dispose()，IDE就会填写这个方法，其中包含一个base.Dispose()调用：

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
}
```

**4 增加代码来撤销定时器。**

在IDE新增的Dispose()方法的最后增加代码，当disposing参数为true时调用animationTimer.Dispose()：

```
protected override void Dispose(bool disposing) {
    base.Dispose(disposing);
    if (disposing) {
        animationTimer.Dispose();
    }
}
```

这里我们覆盖了控件的IDisposable.Dispose()实现要调用的一个protected Dispose()方法。它应当只在disposing参数为true时撤销定时器。

现在BeeControl会在它自己的Dispose()方法中撤销它的定时器。完全自行清理！

**如果一个控件是你自己从头编写的，它要负责撤销它创建的所有其他控件（或可撤销的对象）。**

不过不要只听我们讲，在你增加的那行代码上设置一个断点，运行程序。每次从窗体的Controls集合删除一个BeeControl对象时，就会调用它的Dispose()方法。

**我们不打算再详细讨论这种撤销模式。不过，如果你打算构建定制控件，务必要读读这个文档：**
**http://msdn.microsoft.com/en-us/library/system.idisposable.aspx**

# UserControl是构建控件的简便方法

构建你自己的工具条控件还有一个更容易的方法。不必创建一个继承现有控件的类，只需使用IDE向工程增加一个**UserControl**。对UserControl的处理就像处理窗体一样。可以把任何控件从工具条拖到UserControl上，它也使用IDE中的窗体设计工具。类似于窗体，还可以使用UserControl的事件。所以下面使用UserControl重新构建BeeControl。

动手做

1. 创建一个全新的Windows Forms Application工程。为资源增加4个蜜蜂图片。向窗体拖入一个按钮，仍然使用增加和删除BeeControl同样的代码。

2. 在Solution Explorer中右键点击工程，并选择“Add”→“User Control……”，让IDE增加一个名为**BeeControl**的用户控件。IDE会在窗体设计工具中打开这个新控件。

使用老蜜蜂控件的animationTimer_Tick()方法和cell字段。

3. 将一个Timer控件拖到这个用户控件上。就像把定时器拖到窗体中一样，这个控件会显示在设计工具下方。使用Properties窗口将其命名为**animationTimer**，并将**Interval**设置为150，设置**Enabled**为true。然后双击这个定时器，IDE会增加它的Tick事件处理程序。只需使用前面完成第一个蜜蜂控件动画时使用的Tick事件处理程序。

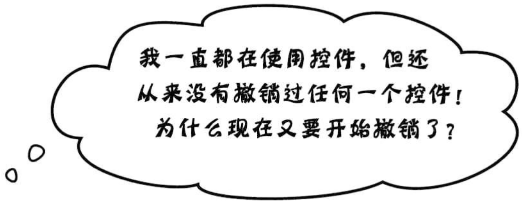
4. 现在更新BeeControl的构造函数：

```
public BeeControl() {
    InitializeComponent();
    BackColor = System.Drawing.Color.Transparent;
    BackgroundImageLayout = ImageLayout.Stretch;
}
```

也可以在IDE的Properties页做这个工作，而不是编写代码来设置。

5. 现在运行程序。按钮代码的表现应该与从前一样。只不过现在它会创建新的基于UserControl的BeeControl。这个按钮可以增加和删除基于UserControl实现的BeeControl。

**UserControl是向工具条增加控件的一种简便方法。编辑UserControl就像编辑窗体一样，可以拖入工具条上的其他控件，而且类似于窗体事件，还可以处理UserControl事件。**






**之所以你没有撤销过控件，那是因为窗体已经为你做了这个工作。**

不过不要只听我们的一面之辞。使用IDE的搜索功能在工程中搜索“Dispose”，你会发现IDE在`Form1.Designer.cs`中增加了一个方法来覆盖`Dispose()`方法，这个方法调用了它自己的`base.Dispose()`。窗体撤销时，它会自动撤销其Controls集合中的所有控件，所以不必你操心。不过，如果要从这个集合删除控件或者在`Controls`集合以外创建控件的新实例（如`BeeControl`中的`Timer`），就需要你自己来完成撤销。

## there are no Dumb Questions

**问：不论使用基于PictureBox的BeeControl还是使用基于UserControl的BeeControl，为什么相应的窗体代码完全一样？**

**答：** 因为这个窗体代码并不关心`BeeControl`对象是如何实现的。它所关心的只是能向窗体的`Controls`集合增加这个对象。

**问：我在Solution Explorer中双击了OldBeeControl，弹出一个有关于向类增加组件的消息，这是什么意思？**

**答：** 向工程增加一个从PictureBox或其他控件继承的类，从而创建一个控件时，IDE会做一些很聪明的事情。其中之一就是允许使用组件，所谓组件（component）是指出现在窗体下方的那些非可视化控件（如Timer和OpenFileDialog）。可以试试看，创建一个继承PictureBox的新类。然后重新构建工程，在IDE中双击这个类，可以得到这样一条消息：

**要向类增加组件，将其从工具条拖出，并使用Properties窗口设置它们的属性。**

把一个OpenFileDialog从工具条拖出，放入这个新类中。它会显示为一个图标。可以点击这个图标来设置它的属性。接下来设置这个控件的一些属性。然后回到类的代码。检查它的构造函数，可以看到，IDE增加了实例化OpenFileDialog对象以及设置其属性的代码。

**问：改变OpenFileDialog的属性时，我注意到IDE中出现一个错误消息“You must rebuild your project for the changes to show up in any open designers”（要在设计工具中显示出所做的改动，必须重新构建工程）。怎么会有这个错误呢？**

**答：** 因为设计工具要运行你的控件，如果没有重新构建代码，它不会运行控件的最新版本。

还记得第一次创建的BeeControl吗？即使只是从工具条将这个控件拖到设计工具中，蜜蜂的翅膀也会扇动。此时还没有运行程序，但是你写的代码正在执行。定时器会触发它的Tick事件，事件处理程序会改变图片。要让IDE反映这种改变，只有一个办法，就是具体编译代码并在内存中运行。所以这会提醒你更新你的代码，从而能适当地显示你的控件。

# 模拟系统的Renderer使用BeeControl在窗体上绘制动画蜜蜂

现在工具已经俱备，可以向模拟系统增加动画了。有了BeeControl类和两个窗体，接下来只需要想办法确定蜜蜂的位置，把蜜蜂从一个地方移到另一个地方，并跟踪这些蜜蜂。还需要确定花在FieldForm上的位置，不过由于花不会移动，所以这很简单。这些代码可以放在一个新的Renderer类中。这个类要完成下面的工作：

我们希望将蜂巢和花场窗体"链接"到统计信息窗体，这很有意义，最小化统计信息窗体时就会最小化蜂巢和花场窗体。为此要告诉Windows：统计信息窗体是它们的父窗体。

稍后我们会构建这个显示工具。不过在具体编写代码之前，先花点时间对Renderer类如何工作做一个计划……

1. **统计信息窗口将作为蜂巢和花场窗体的父窗体。**

向蜂巢模拟系统增加图片的第一步是向工程增加两个窗体。要增加一个名为HiveForm的窗体来显示蜂巢内部，另外增加一个FieldForm窗体（显示有花的花场）。然后向主窗体构造函数增加代码显示这两个子窗体，传入主窗体的引用，告诉Windows，统计信息窗体是它们的父窗体：

```
public Form1() {
    // other code in the Form1 constructor
    hiveForm.Show(this);
    fieldForm.Show(this);
```

每个窗体对象都有一个Show()方法。如果希望把另一个窗体设置为其父窗体，只需将该窗体的一个引用传入Show()。

2. **Renderer保存world和各子窗体的一个引用。**

Renderer类最前面需要一些重要的字段。这个类要知道每个蜜蜂和花的位置，所以需要World的一个引用。而且需要在两个窗体中增加、移动和删除控件，所以还需要各个子窗体的引用：

```
class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;
```

Renderer类最前面需要有这些代码行。这一章的目的就是要补充这个类。

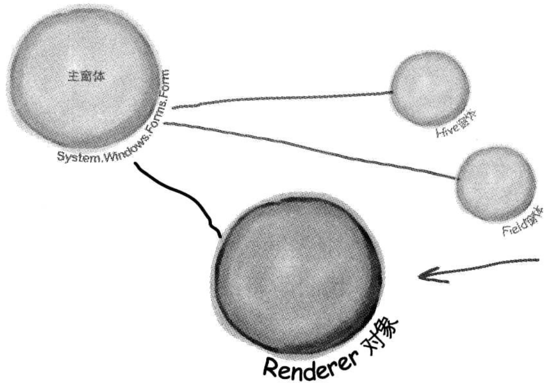
3. **Renderer使用字典来记录控件。**

World使用一个List<Bee>记录Bee对象，并使用一个List<Flower>存储花。Renderer需要能够查看各个Bee和Flower对象，并得出它们分别对应哪个BeeControl和PictureBox控件，如果无法找到对应的控件，就需要创建一个适当的控件。所以这里非常适合使用字典。因此需要在Renderer中增加两个私有字段：

```
    private Dictionary<Flower, PictureBox> flowerLookup =
        new Dictionary<Flower, PictureBox>();
    private Dictionary<Bee, BeeControl> beeLookup =
        new Dictionary<Bee, BeeControl>();
```

这些字典是蜜蜂和花与相应控件之间的一个一一映射。

通过这两个字典集合，Renderer可以为这个世界中的各个蜜蜂或花存储而且只存储一个控件。




**4** **蜜蜂和花已经知道自己的位置。**

使用Point来存储每个蜜蜂和花的位置是有道理的。一旦有一个Bee对象，可以很容易地查找它对应的BeeControl，并设置其位置。

```
beeControl = beeLookup[bee];
beeControl.Location = bee.Location;
```

对于每个蜜蜂或花，可以查找与之对应的控件。然后，将该控件的位置设置为与这个蜜蜂或花对象的位置一致。

**5** **如果一个蜜蜂还没有相应的控件，Renderer会在Hive窗体中为它增加一个控件。**

Renderer要得出一个特定的蜜蜂或花是否有相应控件，这非常容易。对于某个特定的Bee对象，如果字典的`ContainsKey()`方法返回`false`，说明窗体上没有这个蜜蜂对应的控件。所以`Renderer`需要创建一个`BeeControl`，把它增加到字典，然后再把这个控件增加到窗体。（它还会调用控件的`BringToFront()`方法，确保这个控件不会隐藏在花的`PictureBoxes`后面）：

```
if (!beeLookup.ContainsKey(bee)) {
   beeControl = new BeeControl() { Width = 40, Height = 40 };
   beeLookup.Add(bee, beeControl);
   hiveForm.Controls.Add(beeControl);
   beeControl.BringToFront();
} else
   beeControl = beeLookup[bee];
```

ContainsKey()可以告诉我们字典中是否存在这个蜜蜂。如果不存在，就需要增加这个蜜蜂，并增加一个相应的控件。

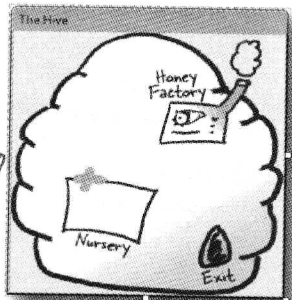
BringToFront()确保在FieldForm中蜜蜂会出现在花的前面（不被花遮挡），如果在HiveForm中则要出现在背景前面。

还记得字典可以使用任意对象作为键吗？不错，这个字典就使用了Bee对象作为键。Renderer需要知道窗体上的哪个BeeControl属于一个特定的蜜蜂。所以它会在字典中查找这个蜜蜂对象，从而得出正确的控件。现在Renderer就可以移动这个控件了。

# 为工程增加蜂巢和花场窗体

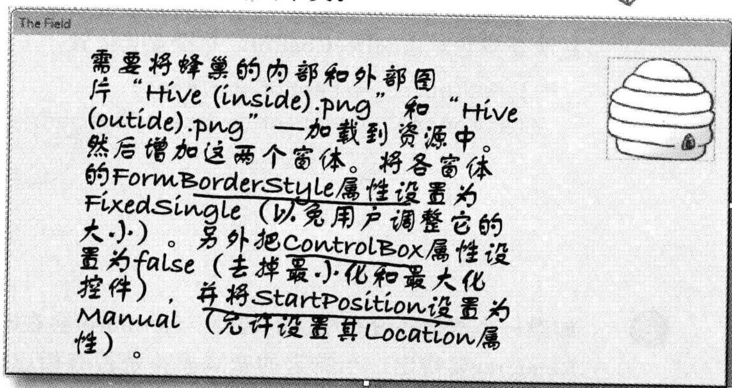
现在需要增加窗体来放置蜜蜂。所以在现有的蜂巢模拟系统工程基础上，使用“Add” “Existing Item……”增加新的**BeeControl**用户控件。UserControl有一个.cs文件、一个.designer.cs文件和一个.resx文件。需要增加这3个文件，然后打开.cs和 .designer.cs文件的代码，修改设置命名空间的代码行，从而与新工程的命名空间一致。重新构建这个工程；BeeControl现在应当会出现在工具条上。还需要向新工程的资源增加图片。 然后再向这个工程增加两个Windows窗体，为此在Solution Explorer中右键点击工程，并从Add菜单选择“Windows Form……”。如果将两个文件命名为HiveForm.cs和FieldForm.cs，IDE会自动地将其Name属性设置为HiveForm和FieldForm。你已经知道了窗体就是对象，所以HiveForm和FieldForm实际上就是另外两个类。

这是一个PictureBox控件，其BackgroundImage设置为蜂巢的外景图片，BackgroundImageLayout设置为Stretch。将蜂巢图片加载到资源设计工具中时，在Properties窗口中点击BackgroundImage旁边的“…”按钮，这些图片就会在资源列表中显示出来。




设置窗体的BackgroundImage属性为蜂巢内景图片，将BackgroundImageLayout属性设置为 Stretch。

要根据这两个切屏图的比例调整两个窗体的大小。

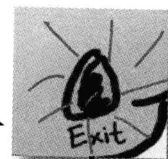


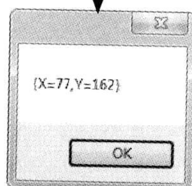

要记住，在Properties窗口中点击闪电图标，打开Events窗口，向下滚动到MouseClick行，并双击这个事件，IDE会为你增加相应的事件处理程序。

## 确定你的位置

需要确定蜂巢位于FieldForm的哪个位置。使用Properties窗口，为Hive窗体的**MouseClick**事件创建一个事件处理程序，并增加以下代码：

```
private void HiveForm_MouseClick(object sender, MouseEventArgs e) {
    MessageBox.Show(e.Location.ToString());
}
```




后面几页会运行窗体。一旦运行，点击图片中蜂巢的出口。事件处理程序会显示所点击位置的具体坐标。

一旦让模拟系统运行，就可以使用这个代码调整Hive的位置集合。

再为FieldForm增加同样的事件处理程序，运行这个窗体。点击这个窗体上的相应位置，得到出口、保育室和蜂蜜工厂的坐标。通过使用所有这些位置，就可以更新上一章所写的Hive类的InitializeLocations()方法：

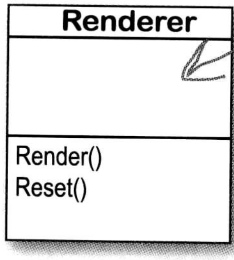
```
private void InitializeLocations()
{
    locations = new Dictionary<string, Point>();
    locations.Add("Entrance", new Point(626, 110));
    locations.Add("Nursery", new Point(77, 162));
    locations.Add("HoneyFactory", new Point(157, 78));
    locations.Add("Exit", new Point(175, 180));}
```

完成后，要删除这个鼠标点击事件处理程序……这个事件处理程序只是用来得到窗体上的位置。

可以直接使用这些坐标，不过，如果你的窗体稍大或稍小，你的坐标可能与此不同。

# 构建Renderer

这是完整的Renderer类。主窗体调用World.Go()在窗体上绘制蜜蜂和花之后，就会调用这个类的Render()方法。除了加载动画蜜蜂图片外，还要确保将花的图片(Flower.png)加载到工程中。




Renderer中的所有字段都是私有的，因为没有其他类需要更新它的属性。Renderer类得到了充分封装。world只需调用Render()在窗体上绘制这个世界，如果需要重置，只需调用Reset()清除窗体上的控件。

```
class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;
```

Renderer保存了world以及要绘制蜜蜂的两个窗体的引用。

```
    private Dictionary<Flower, PictureBox> flowerLookup =
         new Dictionary<Flower, PictureBox>();
    private List<Flower> deadFlowers = new List<Flower>();

    private Dictionary<Bee, BeeControl> beeLookup =
         new Dictionary<Bee, BeeControl>();
    private List<Bee> retiredBees = new List<Bee>();
```

world使用Bee和Flower对象来跟踪这个世界中的各个蜜蜂或花。窗体使用一个PictureBox显示各个花，并使用BeeControl显示每个蜜蜂。Renderer使用这些字典将各个蜜蜂和花连接到它们各自的BeeControl或PictureBox。

花凋谢或蜜蜂退休时，会使用deadFlowers和retiredBees列表来清理字典。

```
    public Renderer(World world, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = world;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
    }

    public void Render() {
        DrawBees();
        DrawFlowers();
        RemoveRetiredBeesAndDeadFlowers();
    }
```

主窗体上运行动画的定时器调用Render()方法，这个方法会更新蜜蜂和花，然后清理字典。

```
    public void Reset() {
        foreach (PictureBox flower in flowerLookup.Values) {
            fieldForm.Controls.Remove(flower);
            flower.Dispose();
        }
        foreach (BeeControl bee in beeLookup.Values) {
            hiveForm.Controls.Remove(bee);
            fieldForm.Controls.Remove(bee);
            bee.Dispose();
        }
        flowerLookup.Clear();
        beeLookup.Clear();
    }
```

如果模拟系统重置，它会各个窗体的Controls.Remove()方法将两个窗体上的控件全部清除。这个方法将查找两个字典中的所有控件，并从窗体上删除，分别调用各个控件的Dispose()。然后将两个字典清空。

用两个foreach循环绘制花。第一个循环查找新花，并增加它们的PictureBox。第二个循环查找死花，删除其PictureBox。

第一个foreach循环使用flowerLookup字典检查每朵花，查看它在窗体上有没有相应的控件。如果没有，则使用一个对象初始化方法创建一个新的PictureBox，把这个控件增加到窗体，然后把它增加到flowerLookup字典。

DrawFlowers()使用Flower对象的Location属性来设置PictureBox在窗体上的位置。

```
private void DrawFlowers() {
    foreach (Flower flower in world.Flowers)
        if (!flowerLookup.ContainsKey(flower)) {
            PictureBox flowerControl = new PictureBox() {
                Width = 45,
                Height = 55,
                Image = Properties.Resources.Flower,
                SizeMode = PictureBoxSizeMode.StretchImage,
                Location = flower.Location
            };
            flowerLookup.Add(flower, flowerControl);
            fieldForm.Controls.Add(flowerControl);
        }

    foreach (Flower flower in flowerLookup.Keys) {
        if (!world.Flowers.Contains(flower)) {
            PictureBox flowerControlToRemove = flowerLookup[flower];
            fieldForm.Controls.Remove(flowerControlToRemove);
            flowerControlToRemove.Dispose();
            deadFlowers.Add(flower);
        }
    }
}
```

第二个foreach循环在flowerLookup字典中查找不应再在窗体上显示的PictureBox，并将其删除。

删除PictureBox后，调用其Dispose()方法。然后将这个Flower对象增加到deadFlowers，以便以后清除。

```
private void DrawBees() {
    BeeControl beeControl;
    foreach (Bee bee in world.Bees) {
        beeControl = GetBeeControl(bee);
        if (bee.InsideHive) {
            if (fieldForm.Controls.Contains(beeControl))
                MoveBeeFromFieldToHive(beeControl);
        } else if (hiveForm.Controls.Contains(beeControl))
            MoveBeeFromHiveToField(beeControl);
        beeControl.Location = bee.Location;
    }

    foreach (Bee bee in beeLookup.Keys) {
        if (!world.Bees.Contains(bee)) {
            beeControl = beeLookup[bee];
            if (fieldForm.Controls.Contains(beeControl))
                fieldForm.Controls.Remove(beeControl);
            if (hiveForm.Controls.Contains(beeControl))
                hiveForm.Controls.Remove(beeControl);
            beeControl.Dispose();
            retiredBees.Add(bee);
        }
    }
}
```

DrawBees()也使用了两个foreach循环，它与DrawFlowers()的做法基本相同。不过，这个方法更复杂一些，所以为便于理解，我们把它的行为分解为两个方法来实现

DrawBees()检查是否有蜜蜂在蜂巢里而它的控件显示在FieldForm上，或者是否有蜜蜂在花场里而相应控件在HiveForm上。这里使用了另外两个方法在两个窗体间移动BeeControl。

一旦删除BeeControl，需要调用其Dispose()方法，这个用户控件会自行撤销它的定时器。

第二个foreach循环的工作与DrawFlowers()中类似，只不过它需要从适当的窗体删除BeeControl。

要保证在Renderer类文件最前面增加using System.Drawing和using System.Windows.Forms 。

GetBeeControl()在beeLookup字典中查找一个蜜蜂，并返回这个蜜蜂。如果没有找到，则创建一个新的BeeControl（大小为40 x 40），并增加到蜂巢窗体（因为蜜蜂都在蜂巢里出生）。

不要忘了，!表示NOT。

```
private BeeControl GetBeeControl(Bee bee) {
    BeeControl beeControl;
    if (!beeLookup.ContainsKey(bee)) {
        beeControl = new BeeControl() { Width = 40, Height = 40 };
        beeLookup.Add(bee, beeControl);
        hiveForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    else
        beeControl = beeLookup[bee];
    return beeControl;
}
```

MoveBeeFromHiveToField()从蜂巢窗体的Controls集合取出一个特定的BeeControl，并把它增加到花场窗体的Controls集合。

```
private void MoveBeeFromHiveToField(BeeControl beeControl) {
    hiveForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(20, 20);
    fieldForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

花场窗体上的蜜蜂比蜂巢窗体上的蜜蜂要小，所以这个方法需要调整BeeControl的Size属性。

```
private void MoveBeeFromFieldToHive(BeeControl beeControl) {
    fieldForm.Controls.Remove(beeControl);
    beeControl.Size = new Size(40, 40);
    hiveForm.Controls.Add(beeControl);
    beeControl.BringToFront();
}
```

MoveBeeFromFieldToHive()将一个BeeControl移回到蜂巢窗体。这里需要再把控件变大（还原为40 x 40）。

```
private void RemoveRetiredBeesAndDeadFlowers() {
    foreach (Bee bee in retiredBees)
        beeLookup.Remove(bee);
    retiredBees.Clear();
    foreach (Flower flower in deadFlowers)
        flowerLookup.Remove(flower);
    deadFlowers.Clear();
}
}
```

一旦DrawBees()和DrawFlowers()发现一朵花或蜜蜂不再在这个世界上，会把它们分别增加到deadFlowers和retiredBees列表中，以便在这一帧的最后将其删除。

所有控件移动之后，Renderer会调用这个方法清除两个字典中已死的花和退休的蜜蜂。

# 现在将主窗体连接到两个新窗体：HiveForm和FieldForm

有一个Renderer固然很棒，但是到目前为止，还没有可以展示控件的窗体。回到主窗体类（可能名为Form1）来修正这个问题，对代码做一些修改：

```
public partial class Form1 : Form {
  private HiveForm hiveForm = new HiveForm();
  private FieldForm fieldForm = new FieldForm();
  private Renderer renderer;

  // the rest of the fields

  public Form1() {
    InitializeComponent();

    MoveChildForms();
    hiveForm.Show(this);
    fieldForm.Show(this);
    ResetSimulator();

    timer1.Interval = 50;
    timer1.Tick += new EventHandler(RunFrame);
    timer1.Enabled = false;
    UpdateStats(new TimeSpan());
  }

  private void MoveChildForms() {
    hiveForm.Location = new Point(Location.X + Width + 10, Location.Y);
    fieldForm.Location = new Point(Location.X,
      Location.Y + Math.Max(Height, hiveForm.Height) + 10);
  }

  public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    renderer.Render();
    // previous code
  }

  private void Form1_Move(object sender, EventArgs e) {
    MoveChildForms();
  }
```

加载主窗体时，它要分别创建这两个窗体的一个实例。它们现在只是堆上的对象，在调用Show()方法之前不会显示。

重置world的代码都将移到ResetSimulator()方法中。

将实例化World的代码移到ResetSimulator()方法中。

窗体把自己的引用传入Form.Show()，从而成为父窗体。

主窗体的构造函数把两个子窗体移到适当的位置，然后显示这两个子窗体。再调用ResetSimulator()，这会完成Renderer的实例化。

由于两个子窗体的StartPosition都设置为Manual，所以主窗体可以使用Location属性移动这两个子窗体。

这个代码会移动两个子窗体，使蜂巢窗体与主统计信息窗体并排放置，花场窗体在这两个窗体下方。

把这行代码增加到RunFrame，使得每次调用world的Go()方法之后模拟系统会更新图序。

每次移动主窗体时会触发Move事件。调用MoveChildForms()确保子窗体会随着主窗体移动。

使用Properties窗口中的Events按钮增加Move事件处理程序。

确保已经将花场和蜂巢窗体的StartPosition属性设置为Manual，否则MoveChildForms()无法工作。

在这里创建World和Renderer类的新实例，这会重置模拟系统。

```
private void ResetSimulator() {
  framesRun = 0;
  world = new World(new Bee.BeeMessage(SendMessage));
  renderer = new Renderer(world, hiveForm, fieldForm);
}

private void reset_Click(object sender, EventArgs e) {
   renderer.Reset();
   ResetSimulator();
   if (!timer1.Enabled)
       toolStrip1.Items[0].Text = "Start simulation";
}

private void openToolStripButton_Click(object sender, EventArgs e) {
  // The rest of the code in this button stays exactly the same.

  renderer.Reset();
  renderer = new Renderer(world, hiveForm, fieldForm);
}
}
```

Reset按钮需要调用Reset()清除所有BeeControl以及花的PictureBox，然后重置模拟系统。

最后，需要为ToolStrip上的Open按钮增加代码，使用Reset()方法从两个窗体的Controls集合删除蜜蜂和花，然后使用新加载的world创建一个新的Renderer。

## there are no Dumb Questions

**问：** 我看到你用一个Show()方法来显示窗体，但是我还是不清楚为什么要传入this作为参数。

**答：** 归根结底，这是因为窗体也只是一个类。显示一个窗体时，只是实例化这个类，并调用它的Show()方法。Show()有一个重载版本，它取一个父窗口作为参数。如果一个窗体是另一个窗体的父窗体，Windows会在它们之间建立一种特殊的关系，例如，最小化父窗口时，会自动最小化这个窗体的所有子窗口。

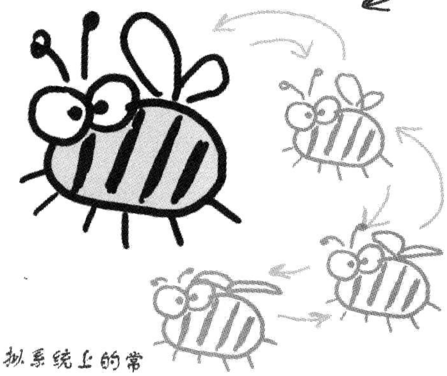
**问：** 能修改现有的控件吗？可以改变内置控件的代码吗？

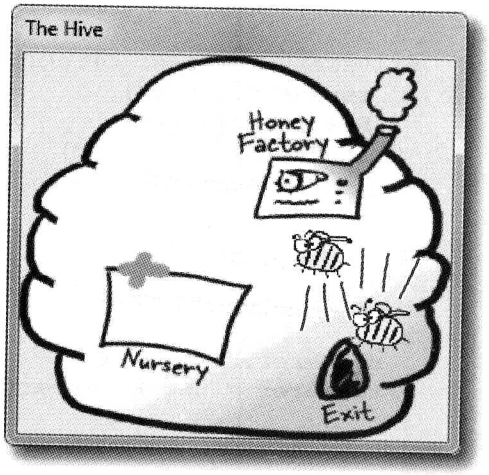
**答：** 不能，实际上根本不能访问Visual Studio内置控件的代码。不过，这些控件都是类，允许你继承，例如继承PictureBox来创建你的BeeControl。如果希望增加或修改某个控件的行为，可以在派生类中增加你自己的方法和属性来处理基类中的成员。

# 运行测试…… 啊咳…… 嗡嗡

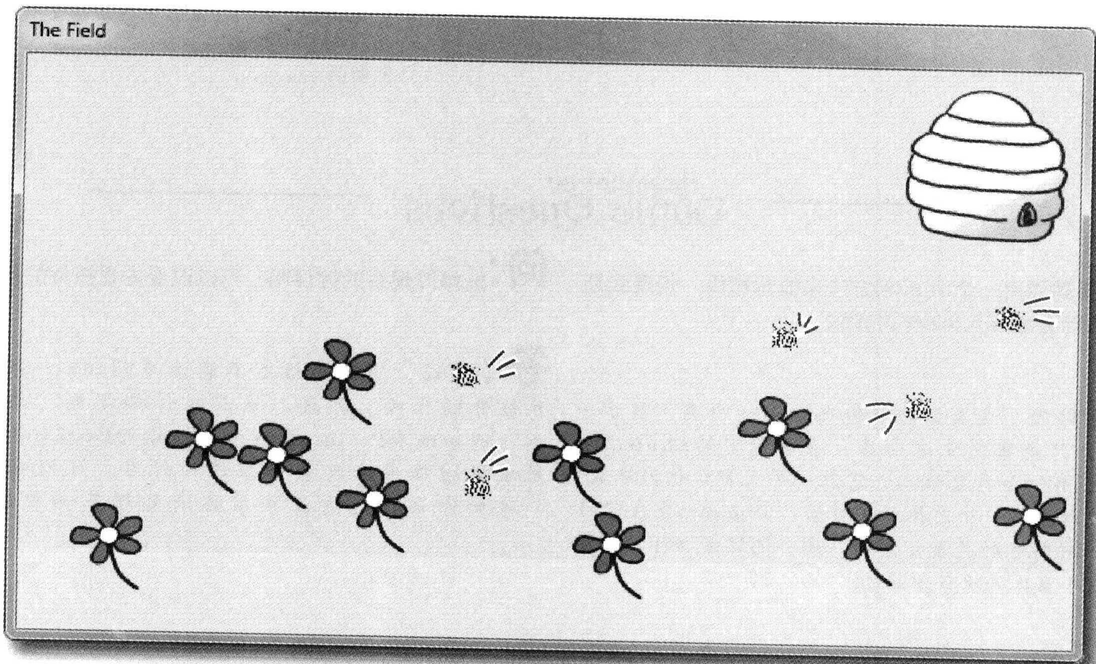
现在编译所有代码，修正可能出现的问题，然后运行你的模拟系统。

蜜蜂现在应该能快乐地扇动翅膀了。






尝试修改模拟系统上的常量，看看Renderer如何处理更多的蜜蜂和花。



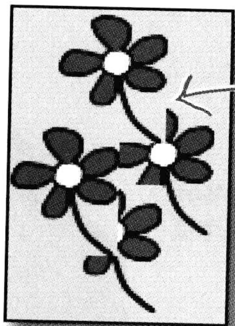
# 看起来不错，不过还有些不太对劲……

仔细看在蜂巢和花中嗡嗡飞舞的蜜蜂，你会注意到这些蜜蜂的显示有点问题。还记得我们已经把各个`BeeControl`的`BackColor`属性设置为`Color.Transparent`，对不对?遗憾的是，这还不足以使这个模拟系统避免图形化程序中相当典型的一些问题。

1. **存在严重的性能问题。**
注意到了吗？所有蜜蜂都在蜂巢里时，整个模拟系统的速度会变慢。如果没有看出速度减慢，可以增大Hive类中的常量来增加更多的蜜蜂试试看。注意帧速率，然后增加更多的蜜蜂，可以看到帧速度开始显著下降。

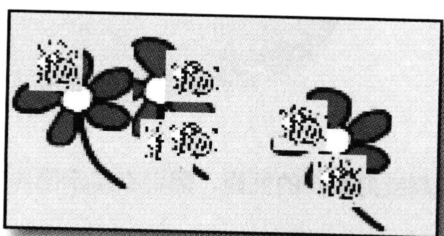
2. **花的背景并不是透明的。**
还有一个与性能无关的问题。保存花的图片文件时，我们提供了透明背景。这样能保证每朵花的背景与窗体的背景一致，但是花相互重叠时看上去就不那么好看了。

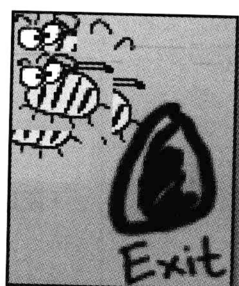
设置一个PictureBox的背景颜色为Transparent时，它会在图像上绘制透明像素，从而与窗体的背景一致……但这不一定正确。



一个PictureBox与另一个PictureBox重叠时，C#会绘制透明像素，使它们与窗体一致，而不是与所覆盖的另一个控件一致，这就会导致两个花重叠时出现这种奇怪的矩形“切角”。

3. **蜜蜂的背景也不是透明的。**
可以看出Color.Transparent确实有一些局限性。蜜蜂在花上停留时，也会出现同样的“切角”问题。对于蜂巢窗体，透明性的处理要好一些，窗体的背景图像没有透过蜜蜂图片的透明区显示出来。但是当蜜蜂重叠时，也会出现同样的问题。如果仔细观察蜜蜂在蜂巢里的移动，会看到蜜蜂图像移动时有时会变形。

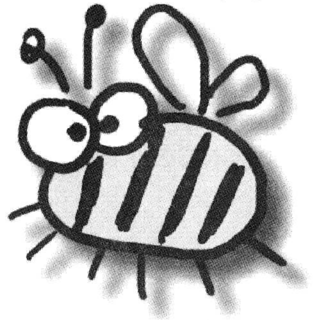




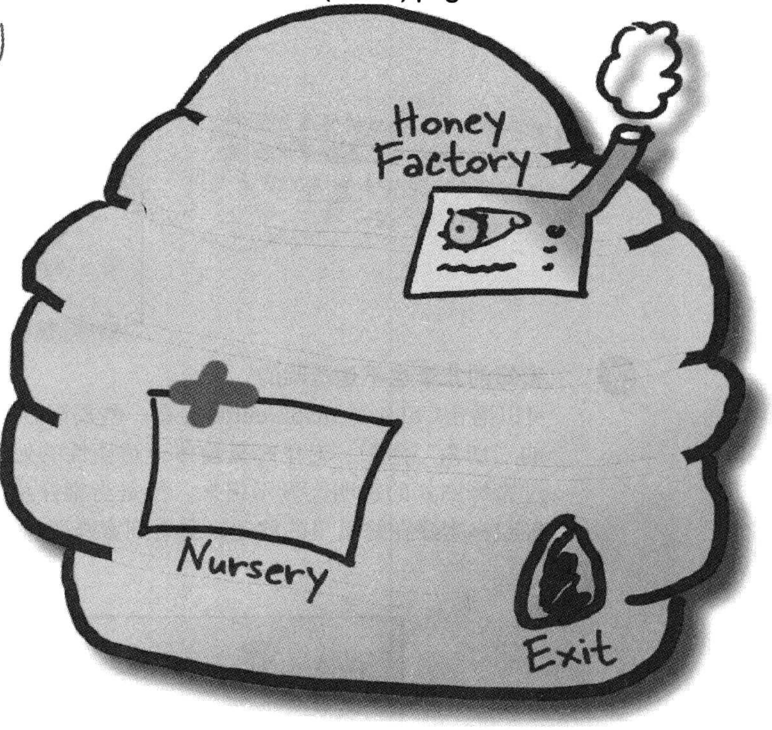
# 仔细考虑性能问题

你下载的每一个蜜蜂图片都很大，确实非常大。可以自己在Windows Picture Viewer中打开其中一个图片看看。既然图片很大，这说明每次PictureBox改变图像时都需要收缩，而放大和缩小图像是需要花费时间的。如果有大量蜜蜂在蜂巢里飞，蜜蜂的移动就会慢得多，造成这种现象的原因就在于蜂巢内部图片过于庞大。将BeeControl的背景设置为透明时，需要做两个工作：首先，必须缩小蜜蜂图片，然后需要缩小窗体背景图片的一部分，使这部分能透过蜜蜂的透明区显示出来。

**Bee animation 1.png**




**Hive (Inside).png**




**蜜蜂的图片文件非常大。每次PictureBox显示一个新的动画帧时，都需要缩小这个图片。这要花费很多时间……**

**……所以，要想提高模拟系统的性能，需要在显示图片之前先缩小所有图片。**

为了提高图形化程序的性能，需要为Renderer增加一个方法，将所有图像收缩至不同的大小。加载图片时就可以调整它的大小，在蜜蜂控件和蜂巢窗体背景上只使用收缩后的版本。

**动手做**

**1 向renderer增加ResizeImage方法。**

工程中的所有图片（如Properties.Resources.Flower）都存储为Bitmap对象。以下是一个调整位图大小的静态方法。把它增加到Renderer类：

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {
    Bitmap resizedPicture = new Bitmap(width, height);
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {
        graphics.DrawImage(picture, 0, 0, width, height);
    }
    return resizedPicture;
}
```

后面几页还会更详细地讨论这个Graphics对象是什么，以及这个方法如何工作。

**2 为BeeControl增加ResizeCells方法。**

BeeControl会存储它自己的Bitmap对象，这是一个包含4个Bitmap对象的数组。以下方法会填充这个数组，按控件大小调整各个位图的大小：

```
private Bitmap[] cells = new Bitmap[4];
private void ResizeCells() {
  cells[0] = Renderer.ResizeImage(Properties.Resources.Bee_animation_1, Width, Height);
  cells[1] = Renderer.ResizeImage(Properties.Resources.Bee_animation_2, Width, Height);
  cells[2] = Renderer.ResizeImage(Properties.Resources.Bee_animation_3, Width, Height);
  cells[3] = Renderer.ResizeImage(Properties.Resources.Bee_animation_4, Width, Height);
}
```

这些代码取各个存储蜜蜂图片的Bitmap对象，使用我们编写的ResizeImage()方法进行收缩。

**3 修改switch语句以使用cells数组，而不是资源。**

BeeControl的Tick事件处理程序有一个switch语句，用于设置它的BackgroundImage：

```
BackgroundImage = Properties.Resources.Bee_animation_1;
```

把Properties.Resources.Bee _ animation _ 1替换为cells[0]。再替换其余各行，使case 2中使用cells[1]，case 3使用cells[2], case 4使用cells[3], case 5使用cells[2],default case使用cells[1]。这样一来只会显示调整大小后的图像。

**4 为BeeControl增加ResizeCells()调用**

需要增加两个ResizeCells()方法调用。首先，在构造函数的最后增加一个ResizeCells()调用。然后在Properites窗口中双击BeeControl，回到IDE设计工具窗口。打开Properties窗口的Events页（点击闪电图标），向下滚动到Resize，再双击这个事件，增加一个Resize事件处理程序。在这个新的Resize事件处理程序中也调用ResizeCells()，这样一来，每次窗体调整大小时也会调整其动画图片。

**5 手动设置窗体的背景图像。**

进入Properties窗口，设置蜂巢窗体的背景图像为 (none)。然后在它的构造函数中将图像设置为适当的大小。

```
public partial class HiveForm : Form {
    public HiveForm() {
        InitializeComponent();
        BackgroundImage = Renderer.ResizeImage(
                Properties.Resources.Hive__inside_,
                ClientRectangle.Width, ClientRectangle.Height);
    }
}
```

窗体有一个ClientRectangle属性，其中包含一个Rectangle，提供了其显示区的大小。

**再来运行模拟系统，现在会快得多！**

# 使用一个Graphics对象调整Bitmap大小

下面来仔细分析为Renderer增加的`ResizeImage()`方法。首先它按指定大小创建一个新的`Bitmap`对象（图片要调整到指定的这个大小）。然后使用`Graphics.FromImage()`来创建一个新的`Graphics`对象。使用这个`Graphics`对象的`DrawImage()`方法将图片绘制到`Bitmap`。注意这里向`DrawImage()`传递了`Width`和`Height`参数，以此告诉它将图像收缩到这个新的大小。最后，返回创建的新`Bitmap`，它可以用作为窗体的背景图像或4个动画图片之一。

窗体和控件都有一个CreateGraphics()方法，可以返回一个新的Graphics对象。稍后还会讨论这个内容。

向这个方法传入一个图片，另外还要传入新的宽度和高度。

```
public static Bitmap ResizeImage(Bitmap picture, int width, int height) {
    Bitmap resizedPicture = new Bitmap(width, height);
    using (Graphics graphics = Graphics.FromImage(resizedPicture)) {
        graphics.DrawImage(picture, 0, 0, width, height);
    }
    return resizedPicture;
}
```

FromImage()方法返回一个新的Graphics对象，可以用它来绘制图像。可以花点时间研究一下，使用IDE的智能提示窗口查看Graphics类的方法。调用DrawImage()时，它会把这个图像复制到resizedPicture的(0, 0)位置，并按width和height参数调整大小。

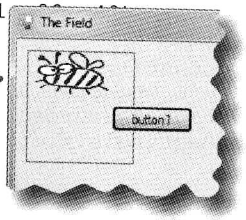
# 来看图像如何调整大小

只是临时性的。尝试后要删除这个按钮和代码。

把一个按钮拖到`Field`窗体上，并增加以下代码。它会创建一个100×100像素的`PictureBox`控件，设置其边框为黑线，以便看出它有多大。然后使用`ResizeImage()`将一个蜜蜂图片压扁到80×40像素，并把这个新图片赋至`PictureBox`控件的`Image`属性。将PictureBox增加到窗体后，就会显示这个蜜蜂。

```
private void button1_Click(object sender, EventArgs e)
{
    PictureBox beePicture = new PictureBox();
    beePicture.Location = new Point(10, 10);
    beePicture.Size = new Size(100, 100);
    beePicture.BorderStyle = BorderStyle.FixedSingle;
    beePicture.Image = Renderer.ResizeImage(
        Properties.Resources.Bee_animation_1
    Controls.Add(beePicture);
}
```

可以看到图像大小的调整，压扁的蜜蜂图像比PictureBox小得多。ResizeImage()把它压扁了。



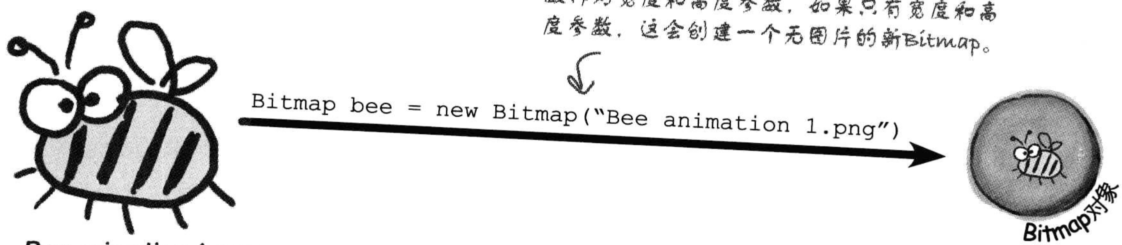

**ResizeImage()方法创建一个Graphics对象，可以在一个不可见的Bitmap对象上绘图。这个方法将返回这个Bitmap，以便在窗体或PictureBox中显示。**

# 图像资源存储在Bitmap对象中

将图片文件导入工程资源中时，到底发生了什么？你已经知道可以使用Properties.Resources来访问这些资源。但是，导入后程序到底对这些资源做了什么处理？

.NET 会把图像转变为一个新的Bitmap对象：

Bitmap类有多个重载构造函数。这个构造函数从磁盘加载一个图片。另外还可以传入整数作为宽度和高度参数，如果只有宽度和高度参数，这会创建一个无图片的新Bitmap。




Bee animation 1.png

## 将各个Bitmap绘制到屏幕上

图像导入到Bitmap对象后，窗体可以利用如下调用把它们绘制到屏幕上：

这个调用得到一个Graphics对象，可以在窗体上绘图。我们使用了一个using语句，以确保撤销这个Graphics对象。

```
using (Graphics g = CreateGraphics()) {
    g.DrawImage(myBitmap, 30, 30, 150, 150);
}
```

DrawImage()取一个Bitmap参数，这是要绘制的图像……

……起始X，Y坐标……

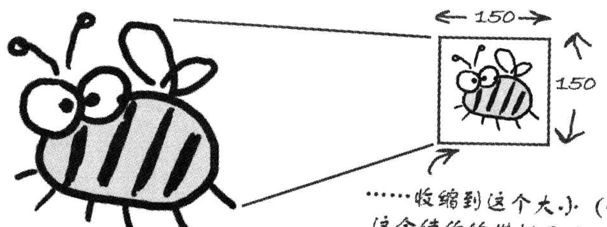
……以及一个大小（150x150像素）。

## 图像越大……

注意到DrawImage()的最后两个参数吗?如果Bitmap中的图像大小是175 × 175呢?图形库必须把图像大小调整为150 × 150。如果Bitmap包含一个1500 × 2025的图像呢?大小调整就会更慢……

**调整图像大小很占用处理时间！如果只做一次，无关大碍。但是，如果每个时间帧都要调整大小，程序运行就会减慢。我们为蜜蜂和蜂巢提供的图片非常大。Renderer移动蜜蜂时（特别是在蜂巢内部图片前移动蜜蜂），必须反复地调整大小。这就导致了性能问题！**

这个图像大小为300x300像素……




……收缩到这个大小（例如150x150像素）。这会使你的模拟系统速度减慢！

# 使用System.Drawing自行控制图形化

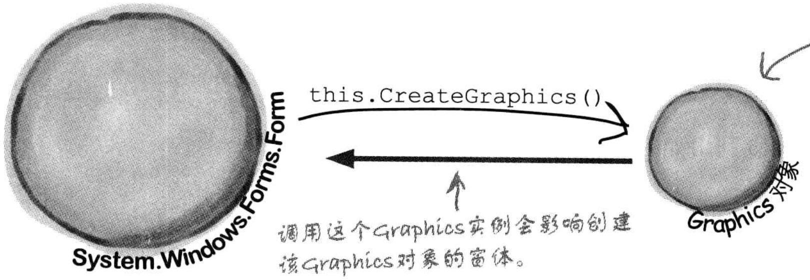
Graphics对象属于System.Drawing命名空间。.NET Framework提供了一些非常强大的图形化工具，它们的功能绝不只是工具条中的PictureBox控件那么简单。利用这些工具，可以绘制形状、使用字体，完成各种复杂的图形化操作……所有这些都从一个Graphics对象开始。如果想增加或修改任何对象的图形或图像，由于要在这个对象上绘图，可以创建与该对象相关联的一个Graphics对象，再使用这个Graphics对象的方法在这个目标对象上绘图。

**System.Drawing**

System.Drawing命名空间中的图形化方法有时称为GDI+，这表示图形化设备接口（Graphics Device Interface）。用GDI+绘制图形时，首先要有一个Graphics对象，它关联到一个Bitmap、窗体、控件或其他目标对象（希望使用Graphics对象的方法在这个对象上绘图）。

**1 首先考虑绘图的目标对象。**

例如，考虑一个窗体。调用窗体的CreateGraphics()方法时，它会返回为了在窗体上绘图而创建的一个Graphics实例。



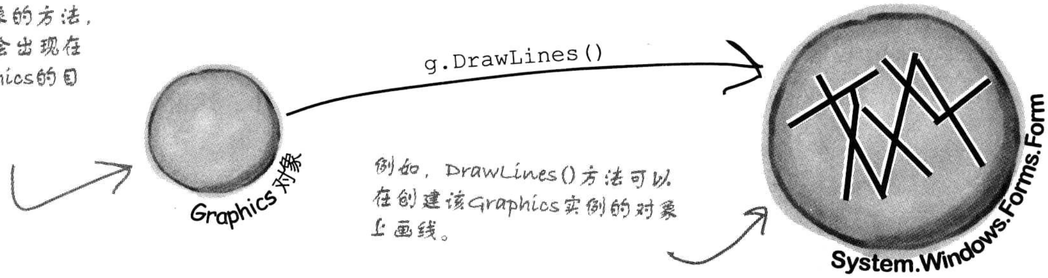

窗体可以调用它自己的CreateGraphics()方法，或者其他对象也可以调用这个方法。无论以何种方式调用，都会返回一个Graphics对象的引用，可以使用这个Graphics对象的方法在窗体上绘图。

不会在Graphics对象自身上绘图。只是使用它在其他对象上绘图。

**2 使用Graphics对象的方法在目标对象上绘图。**

每个Graphics对象都提供了很多方法，可以在创建该Graphics对象的目标对象上绘图。调用这个Graphics对象的方法来画线、圆、矩形、文本以及图像时，这些图形会出现在窗体上。

尽管调用的是Graphics对象的方法，但具体图形会出现在创建该Graphics的目标对象上。



如果类中要使用这些方法，一定要在类前面增加using System.Drawing;。或者，向工程增加一个窗体时，IDE会自动向窗体增加这行代码。

# GDI+图形化30秒之旅

一旦创建了Graphics对象，可以绘制各种图形和图片。只需调用它的方法，就能直接在创建该Graphics实例的对象上绘图。

1. 第一步往往是得到一个Graphics对象。可以使用窗体的CreateGraphics()方法，或者传入一个Graphics对象。要记住，Graphics实现了IDisposable接口，所以，如果创建一个新的Graphics对象，一定要使用using语句：

```
using (Graphics g = this.CreateGraphics()) {
```

记住，这会在创建该实例的对象上绘图。

2. 如果想画一条线，可以调用DrawLine()，并指定起点和终点，分别用X和Y坐标表示：

```
g.DrawLine(Pens.Blue, 30, 10, 100, 45);
```

起点坐标……

……终点坐标。

或者可以使用一对Point来表示：

```
g.DrawLine(Pens.Blue, new Point(30, 45), new Point(100, 10));
```

3. 以下代码绘制一个填充的蓝灰色矩形，然后指定天蓝色边框。它使用一个Rectangle来定义大小，这个矩形的左上角在(150, 15),宽度为140像素，高度为90像素。

```
g.FillRectangle(Brushes.SlateGray, new Rectangle(150, 15, 140, 90));
g.DrawRectangle(Pens.SkyBlue, new Rectangle(150, 15, 140, 90));
```

有很多颜色可供使用，只需键入“Color”，“Pens”或“Brushes”，后面加一个点号，智能提示窗口就会显示出所有颜色。

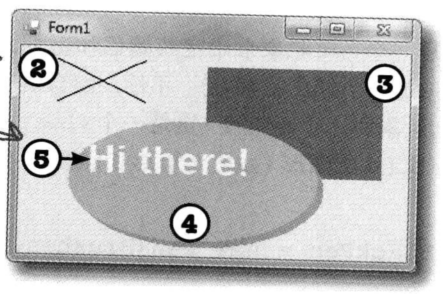
4. 可以使用DrawCircle()或FillCircle()方法绘制椭圆和圆，它们也使用一个Rectangle来指定这个图形应当有多大。以下代码会绘制两个椭圆，稍稍有点偏移，产生一种阴影的效果：

```
g.FillEllipse(Brushes.DarkGray, new Rectangle(45, 65, 200, 100));
g.FillEllipse(Brushes.Silver, new Rectangle(40, 60, 200, 100));
```

5. 使用DrawString()方法用某种字体和颜色绘制文本。为此，需要创建一个Font对象。它实现了IDisposable，所以要使用一个using语句：

```
using (Font arial24Bold = new Font("Arial", 24, FontStyle.Bold)) {
   g.DrawString("Hi there!", arial24Bold, Brushes.Red, 50, 75);
}
```

如果按顺序执行前面的语句，这就是最后得到的窗体。以上的每个语句都与这里的一个标号对应。左上角坐标为(0,0)。




这个图上没有步骤1，因为第一步只是创建Graphics对象。

# 使用Graphics在窗体上绘制图片

下面创建一个新的Windows应用，点击窗体时在窗体上绘制一个图片。

动手画

1 **首先向窗体增加一个Click事件。**

在Properties窗口中打开Events页（点击闪电图标），向下滚动到Click事件，双击这个事件。

事件处理程序最前面是一个创建`Graphics`对象的`using`代码行。使用GDI+时，会使用大量实现IDisposable接口的对象。如果这些对象没有撤销，它们就会慢慢地耗尽计算机的资源，直到你退出程序。所以可能需要使用大量`using`语句：

```
using (Graphics g = CreateGraphics()) {
```

这是Form1_Click()事件处理方法中的第一行代码。我们将提供事件处理程序中的所有代码，把它们集成在一起就能绘制图片。

2 **注意在窗体上绘图的顺序。**

我们希望这个图片有一个天蓝色背景，所以首先画一个很大的蓝色矩形，在此之后绘制的任何对象都会画在这个矩形上面。要充分利用窗体的一个属性：`ClientRectangle`。这是一个`Rectangle`，定义了窗体绘制区的边界。`Rectangle`很有用，可以指定左上角坐标，并指定其宽度和高度来创建一个新矩形。一旦创建一个矩形，它会自动计算`Top`，`Left`，`Right`和`Bottom`属性。而且Rectangle还有一些很有用的方法，如**`Contains()`**，如果一个给定的点在这个矩形内，这个方法就会返回true。

```
g.FillRectangle(Brushes.SkyBlue, ClientRectangle);
```

这本书后面会大量用到这个方法！你认为利用Contains()能做什么？

3 **绘制蜜蜂和花。**

你已经很清楚`DrawImage()`方法如何工作。一定要增加图像资源。

```
g.DrawImage(Properties.Resources.Bee_animation_1, 50, 20, 75, 75);
g.DrawImage(Properties.Resources.Flower, 10, 130, 100, 150);
```

Pen用于画线，而且有宽度。如果想画一个填充的形状或某些文本，就需要一个Brush。

4 **增加一个用来绘图的画笔。**

每次要画线时，就要使用一个`Pen`对象来确定其颜色和粗细。有一个内置的`Pens`类能提供很多画笔（例如，`Pens.Red`是一个红色的细笔）。不过，也可以使用`Pen`类的构造函数创建你自己的画笔，它取一个`Brush`对象和一个粗细度（这是一个`float`，所以最后要有一个`F`）作为参数。画刷可以用来绘制填充（实心）的图形（如填充矩形和椭圆），有一个`Brushes`类可以提供不同颜色的画刷。

```
using (Pen thickBlackPen = new Pen(Brushes.Black, 3.0F)) {
```

这要放在创建Pen的内层using语句中。

5 **增加一个箭头指向花。**

有些Graphics方法取一个Points数组作为参数，并使用一系列直线或曲线把它们连接起来。我们将使用DrawLines()方法画一个箭头，并使用DrawCurve()方法画它的箭柄。还有另外一些方法也取点数组作为参数（如DrawPolygon()，这会画一个封闭的形状，FillPolygon()会画一个填充的封闭形状）。

```
g.DrawLines(thickBlackPen, new Point[] {
    new Point(130, 110), new Point(120, 160), new Point(155, 163)});
g.DrawCurve(thickBlackPen, new Point[] {
    new Point(120, 160), new Point(175, 120), new Point(215, 70) });
}
```

using块到这里结束，由于不再需要thickBlackPen，所以它将撤销。

向DrawCurve()传入一个点数组时，它会画一个平滑的曲线，将这些点按顺序连接起来。

6 **增加一个字体来绘制文本。**

绘制文本时，首先需要创建一个Font对象。这里再一次使用一个using语句，因为Font实现了IDisposable。创建字体很简单。它有很多重载构造函数，最简单的一个构造函数取一个字体名、字体大小和FontStyle enum作为参数。

```
using (Font font = new Font("Arial", 16, FontStyle.Italic)) {
```

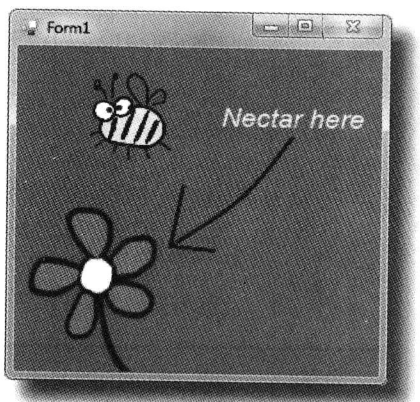
7 **增加文本“Nectar here”。**

既然已经有了字体，下面可以确定在哪里放置文本，为此要度量文本绘制后的大小。MeasureString()方法返回一个定义其大小的SizeF(SizeF就是Size的一个float版本，Size和SizeF都只是定义一个宽度和一个高度）。由于我们知道箭头在哪里结束，所以使用串度量来确定串的中心位于箭头上方。

```
        SizeF size = g.MeasureString("Nectar here", font);
        g.DrawString("Nectar here", font, Brushes.Red, new Point(
            215 - (int)size.Width / 2, 70 - (int)size.Height));
    }
}
```

要保证结束两个using块。

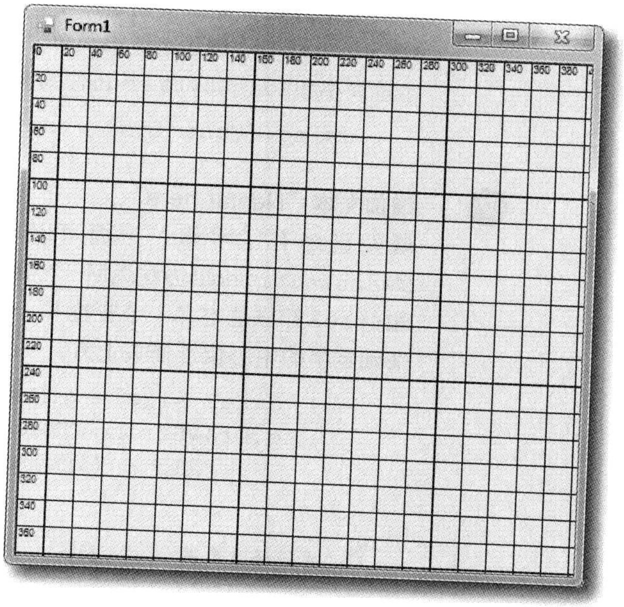
可以指定一个点和一个Size（或宽度及高度）来创建一个Rectangle。一旦创建了矩形，可以得到它的边界，并使用Contains()方法查看是否包含某个Point。



## Sharpen your pencil

1. 处理Graphics时，大多会把窗体想成是一个有X，Y坐标的网格。以下代码会构建下面所示的网格，你的任务填入这里缺少的代码。

```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
  for (int x = 0; x < this.Width; x += 20) {
     ..............................................................
  } ...............................................................
  for (int y = 0; y < this.Height; y += 20) {
     ..............................................................
  } ...............................................................
}
```



2. 运行下面的代码会发生什么？画出在窗体上的输出，使用前面所画的网格来确定特定点的位置。

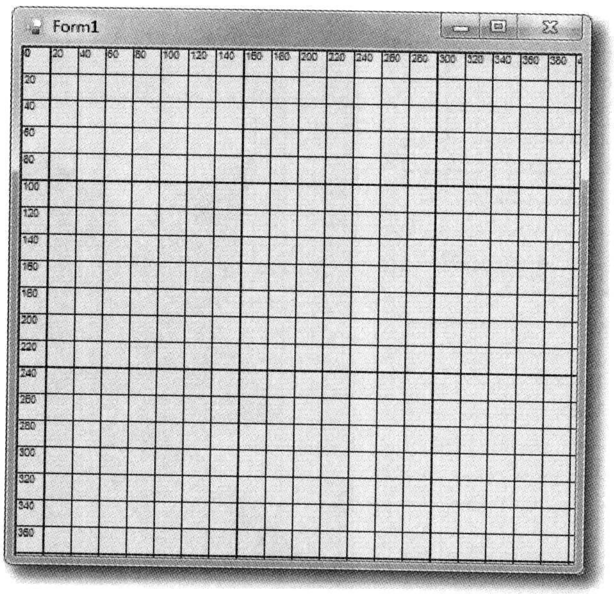
```
using (Pen pen =
        new Pen(Brushes.Black, 3.0F)) {
  g.DrawCurve(pen, new Point[] {
     new Point(80, 60),
     new Point(200,40),
     new Point(180, 60),
     new Point(300,40),
  });
  g.DrawCurve(pen, new Point[] {
     new Point(300,180), new Point(180, 200),
     new Point(200,180), new Point(80, 200),
  });
  g.DrawLine(pen, 300, 40, 300, 180);
  g.DrawLine(pen, 80, 60, 80, 200);
  g.DrawEllipse(pen, 40, 40, 20, 20);
  g.DrawRectangle(pen, 40, 60, 20, 300);
  g.DrawLine(pen, 60, 60, 80, 60);
  g.DrawLine(pen, 60, 200, 80, 200);
}
```

FillPolygon()、DrawLines()和其他一些图形化方法都取一个Point数组作为参数，这个数组定义了一系列相连线段的顶点。

3. 这是处理不规则形状的一些图形化代码。使用下面给出的网格确定会画出什么图形。

```
g.FillPolygon(Brushes.Black, new Point[] {
        new Point(60,40), new Point(140,80), new Point(200,40),
        new Point(300,80), new Point(380,60), new Point(340,140),
        new Point(320,180), new Point(380,240), new Point(320,300),
        new Point(340,340), new Point(240,320), new Point(180,340),
        new Point(20,320), new Point(60, 280), new Point(100, 240),
        new Point(40, 220), new Point(80,160),
    });

using (Font big = new Font("Times New Roman", 24, FontStyle.Italic)) {
    g.DrawString("Pow!", big, Brushes.White, new Point(80, 80));
    g.DrawString("Pow!", big, Brushes.White, new Point(120, 120));
    g.DrawString("Pow!", big, Brushes.White, new Point(160, 160));
    g.DrawString("Pow!", big, Brushes.White, new Point(200, 200));
    g.DrawString("Pow!", big, Brushes.White, new Point(240, 240));
}
```



# Sharpen your pencil Solution

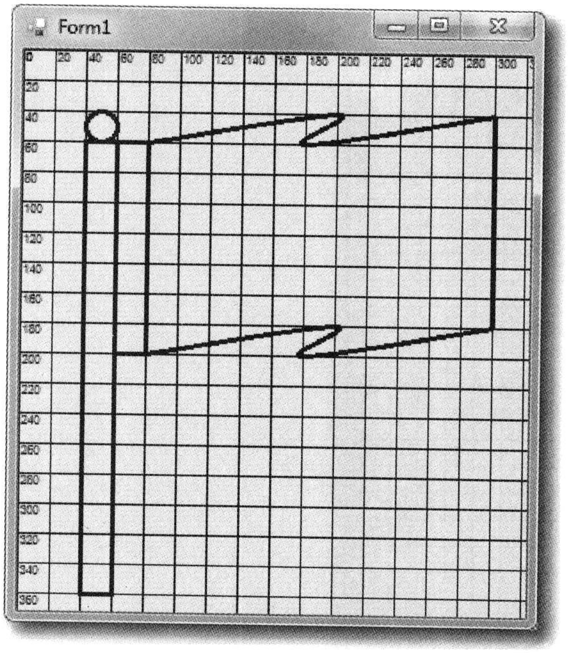
你的任务是填入这里所缺的代码，画一个网格，并在这个网格上画出两段代码生成的图。

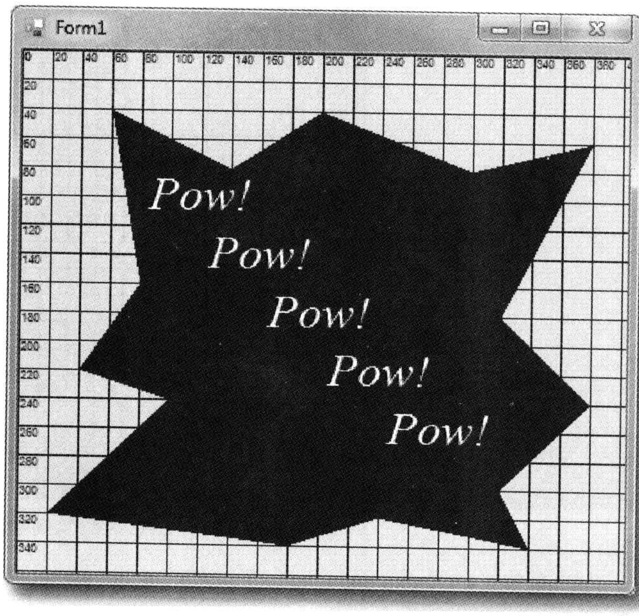
```
using (Graphics g = this.CreateGraphics())
using (Font f = new Font("Arial", 6, FontStyle.Regular)) {
    for (int x = 0; x < this.Width; x += 20) {
        g.DrawLine(Pens.Black, x, 0, x, this.Height);
        g.DrawString(x.ToString(), f, Brushes.Black, x, 0);
    }
    for (int y = 0; y < this.Height; y += 20) {
        g.DrawLine(Pens.Black, 0, y, this.Width, y);
        g.DrawString(y.ToString(), f, Brushes.Black, 0, y);
    }
}
```

首先画出垂直线，以及Y轴旁的数字。沿X轴每20个像素有一条垂直线。

使用using语句确保Graphics和Font对象在窗体完成绘制后撤销。

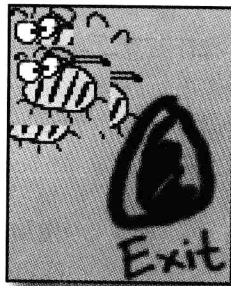
接下来画水平线和X轴上的数。画水平线时，要选择一个Y值，从窗体左边(0, y)到窗体右边(0, this.Width)画一条线。



# Graphics可以修正透明问题……

Renderer绘制蜜蜂时，如果蜜蜂重叠看上去会很怪异。



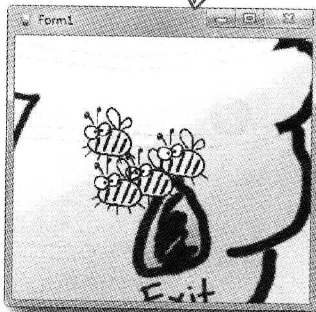

还记得那些讨厌的图形化问题吗？下面就来解决这些问题！DrawImage()正是解决Renderer中透明问题的关键，绘制图像时会在蜜蜂和花周围绘制方框，就导致了重叠问题。先回到前面包含图片的Windows应用，修改应用让它能够绘制相互重叠的大量蜜蜂而不存在图形化问题。

**动手做**

1. 增加一个DrawBee()方法在任意Graphics对象上画一个蜜蜂。它使用了一个重载DrawImage()方法，其中取一个Rectangle确定在哪里绘制这个图像以及图像有多大。

```
public void DrawBee(Graphics g, Rectangle rect) {
  g.DrawImage(Properties.Resources.Bee_animation_1, rect);
}
```

好多了，点击窗体，现在蜜蜂重叠就正常了。



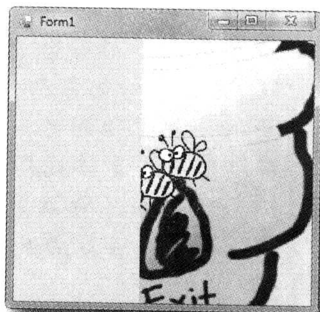

2. 以下是窗体新的Click事件处理程序。仔细看它是如何工作的，绘制蜂巢时，使其左上角偏离窗体，位于（-Width, -Height），绘制的大小是窗体高度和宽度的两倍，这样就能调整窗体的大小，而它仍能正常绘制。然后使用DrawBee()方法画4个蜜蜂。

```
private void Form1_Click(object sender, EventArgs e) {
    using (Graphics g = CreateGraphics()) {
        g.DrawImage(Properties.Resources.Hive__inside_,
            -Width, -Height, Width * 2, Height * 2);
            Size size = new Size(Width / 5, Height / 5);
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 50, Height / 2 - 40), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 20, Height / 2 - 60), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 80, Height / 2 - 30), size));
            DrawBee(g, new Rectangle(
                    new Point(Width / 2 - 90, Height / 2 - 80), size));
    }
}
}
```

首先画蜂巢背景，其左上角与页面有很大偏差，我们只能看到它的一小部分。然后画出4个蜜蜂，让它们相互重叠，如果不重叠，把窗体拉大，再点击窗体使之重叠。

## ……但是有一个问题

不过，如果把它拖出屏幕边沿再拖回来，看看会发生什么！唉呀，不得了！




3. 运行程序，点击窗体，观察它怎样绘制蜜蜂！不过这里还是有问题。如果把窗体拖出屏幕边沿后再拖回来，图片会消失！再来检查前几页写的“Nectar here”程序，它也存在同样的问题！

你觉得为什么会出现这个问题？

**窗体和控件有一个Paint事件，可以提供一个Graphics对象。用它画的任何图形都会自动重绘。**

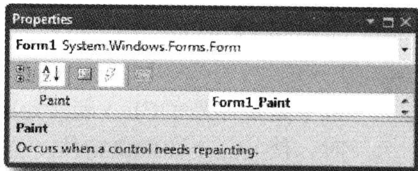
# 使用Paint事件固定图形

一旦窗体的某一部分被覆盖，图片就会从窗体上消失，既然如此，图形化还有什么意义呢？一点用都没有。好在，有一个简单办法可以确保图片留在窗体上：只需编写一个Paint事件处理程序。每次窗体需要重绘时（比如拖出屏幕）就会触发一个Paint事件。它的PaintEventArgs参数有一个Graphics属性，这是一个Graphics对象，用它画的所有内容将“固定”，不再消失。

**1 增加一个Paint事件处理程序。**

在Properties窗口的Events页中双击“Paint”，增加一个Paint事件处理程序。只要窗体上的图形变“脏”就会触发Paint事件。所以在这个事件处理程序中绘制图形就能让这些图像固定。

双击Paint增加一个Paint事件处理程序。它的PaintEventArgs有一个属性Graphics，用这个Graphics对象绘制的任何图形都会保留在窗体上。




**2 使用Paint事件PaintEventArgs参数的Graphics对象。**

不用以using语句开头，这个事件处理程序最前面使用以下代码：

```
private void Form1_Paint(object sender, PaintEventArgs e) {
    Graphics g = e.Graphics;
```

不必使用using语句，因为你没有创建Graphics对象，所以不用撤销。

**3 复制绘制重叠蜜蜂和蜂巢的代码**

将上一页的DrawBees()方法增加到新的用户控件。然后将Click事件的代码复制到新的Paint事件处理程序中，不过不要复制第一行**using**语句，因为已经有一个名为g的Graphics对象（由于没有using语句，所以一定要去掉它相应的结束大括号）。现在运行这个程序。图片将固定！

对绘制“Nectar here”文本的代码做同样的处理，让这个文本也固定在窗体上。

## 窗体和控件一直在自行重绘

尽管可能看不出来，但是你的窗体确实必须一直自行重绘。只要窗体上有控件，它们就会显示图形，标签会显示文本、按钮显示一个按钮图片，复选框显示一个小方框，其中包含一个x。可以把它们作为控件拖动，但实际上每个控件都在绘制自己的图像。一旦把窗体拖出屏幕或者拖到另一个窗体下面，然后再把它拖回来或者其他窗体不再覆盖它，原先被覆盖的那部分窗体现在就会无效，这说明它不再显示本该显示的图像。此时.NET会向窗体发送一个消息，告诉它自行重绘。只要窗体“变脏”需要重绘时就会触发Paint事件。如果希望你的窗体或用户控件自行重绘，可以通过调用其Invalidate()方法告诉.NET把它“弄脏”。

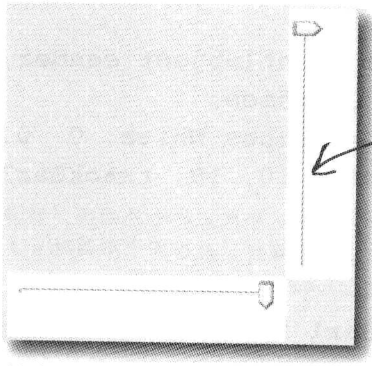
**Exercise**

结合窗体和用户控件的有关知识，看看能不能使用Bitmap对象和DrawImage()方法做些练习，构建一个用户控件，它使用TrackBar控件放大和缩小一个图像。

**1 向一个新用户控件增加两个TrackBar控件。**

创建一个新的Windows Application工程。增加一个用户控件，命名为Zoomer，并设置其Size属性为（300，300）。将两个TrackBar控件从工具条拖到这个用户控件上。把trackBar1拖到这个用户控件下方。然后把trackBar2拖到用户控件的右侧，并设置其Orientation属性为Vertical。两个控件的Minimum属性都设置为1，Maximum设置为175，Value设置为175，TickStyle设置为None。将两个TrackBar的背景颜色设置为白色。最后，分别双击这两个trackbar，增加一个Scroll事件处理程序。让两个事件处理程序调用用户控件的Invalidate()方法。

**用户控件有一个Paint事件，其做法与窗体的Paint事件类似。只需使用这个事件的PaintEventArgs参数e。它有一个属性Graphics，用这个Graphics对象绘制的任何图形会绘制到从工具条拖出的用户控件实例上。**

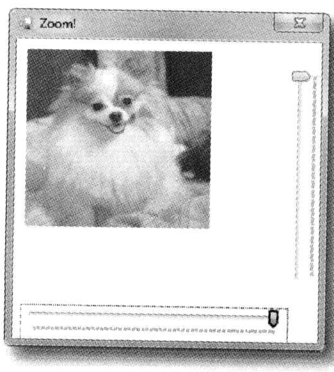


为两个trackbar指定白色背景，这是因为所有图形要画在一个白色矩形上，希望它们能协调。

**2 将图片加载到一个Bitmap对象，并在这个用户控件上画出。**

为Zoomer用户控件增加一个私有Bitmap字段，名为photo。创建Bitmap的实例时，可以使用其构造函数加载你喜欢的图像文件，我们使用了一个毛茸茸的小狗的图片。然后为这个用户控件增加一个Paint事件。Paint事件处理程序应当创建一个Graphics对象，以便在这个控件上绘图。先在整个控件上画一个白色的填充矩形，然后使用DrawImage()将photo字段的内容画在控件上，使其左上角位于(10, 10),宽度为trackBar1.Value，高度为trackBar2.Value。然后把这个控件拖到窗体上，调整窗体的大小，使trackbar恰好在边上。

移动trackbar时，图片会收缩和拉伸！




**用户滚动其中任何一个TrackBar时，会调用用户控件的Invalidate()方法。这会导致用户控件触发Paint事件，调整照片的大小。要记住，由于没有创建Graphics对象（这是在PaintEventArgs参数中传入的），所以不需要撤销这个对象。因此，不必对它使用using语句。只需在Paint事件处理程序中绘制图像。**

## Exercise Solution

使用Bitmap对象和DrawImage()方法做些练习，构建一个窗体，使用Bitmap对象和DrawImage()方法从文件加载一个图片，并放大和缩小。

```
public partial class Zoomer : UserControl {

  Bitmap photo = new Bitmap("c:\Graphics\fluffy_dog.jpg");

  public Zoomer() {
     InitializeComponent();
  }

  private void Zoomer_Paint(object sender, PaintEventArgs e) {
      Graphics g = e.Graphics;
      g.FillRectangle(Brushes.White, 0, 0, Width, Height);
      g.DrawImage(photo, 10, 10, trackBar1.Value, trackBar2.Value);
  }

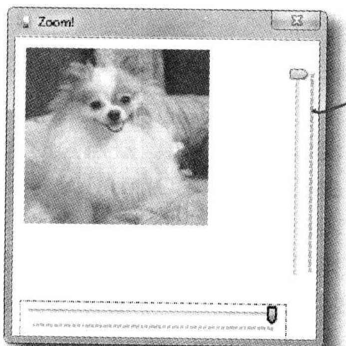
  private void trackBar1_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
  private void trackBar2_Scroll(object sender, EventArgs e) {
     Invalidate();
  }
}
```

这个Bitmap构造函数从一个文件加载图片。Bitmap还有其他重载构造函数，其中一个构造函数允许只指定宽度和高度，这会创建一个空位图。

可以换成你自己的文件，Bitmap构造函数可以使用多种文件格式。更棒的是，可以使用一个OpenFileDialog来放大你指定的任何图像！

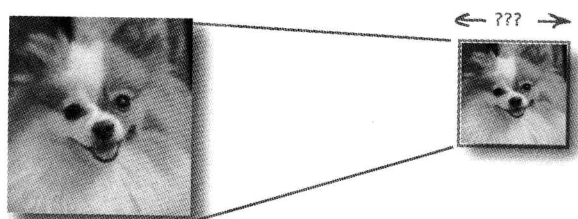
首先绘制一个很大的白色矩形，填满整个控件，接下来会在这个白色矩形上面绘制照片。后面两个参数确定了所绘制图像的大小，trackBar1设置宽度，trackBar2设置高度。

每次用户滑动某个trackbar控件时，会触发一个Scroll事件。让这个事件的事件处理程序调用用户控件的Invalidate()方法，这就会让窗体自行重绘……此时将绘制图像的一个不同大小的新副本。



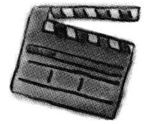

每次拖动这里会导致调用DrawImage()，从而再次调整图像的大小。

```
g.DrawImage(myBitmap, 30, 30, 150, 150);
```



# 进一步分析窗体和控件如何自行重绘

**在幕后**



应该记得，之前我们说过，如果开始使用Graphics对象，那么所有图形化工作都将由你控制。这就像在告诉.NET，“嘿，我知道我在做什么。我可以全权负责。”对于绘制和重绘，你可能不希望窗体在最小化和最大化时重绘…… 或者也可能希望能更频繁地重绘。一旦知道在后台窗体或控件是如何处理的，你就能自己来控制重绘：

1. **每个窗体有一个Paint事件，会在窗体上绘制图形。**

   查看任何一个窗体的事件列表，查找名为**Paint**的事件。只要窗体必须自行重绘，就会触发这个事件。每个窗体和控件在内部都使用一个Paint事件来确定何时重绘。但是谁来触发这个事件呢？它由一个名为**OnPaint**的方法调用，这个方法是窗体或用户控件从`Control`类继承的。（这个方法的命名遵循第11章介绍的命名模式，为触发事件的方法命名时，最前面是“On”，后面是事件名）。打开任意一个窗体，覆盖`OnPaint`：

覆盖任意窗体的OnPaint，增加这样一行代码。

类似于前面对Dispose()的处理。

```
protected override void OnPaint(PaintEventArgs e) {
    Console.WriteLine("OnPaint {0} {1}", DateTime.Now, e.ClipRectangle);
    base.OnPaint(e);
}
```

   拖动窗体，把它的一半拖出屏幕，最小化，或者隐藏在其他窗口下面。仔细查看所写的输出。你会看到，只要窗体的某一部分“变脏”或无效，`OnPaint`方法就会触发`Paint`事件，要求重绘。仔细观察`ClipRectangle`，可以看到这个矩形描述了需要重绘窗体的哪一部分。这个矩形会传递到`Paint`事件的`PaintEventArgs`，因此可以只重绘无效的部分来改善性能。

2. **Invalidate()控制何时重绘，以及重绘什么内容。**

   如果窗体中某些部分出现重叠、覆盖或者移出屏幕，再次显示时，.NET会触发Paint事件。它调用`Invalidate()`，并向这个方法传入一个`Rectangle`。这个`Rectangle`告诉`Invalidate()`方法需要重绘窗体的哪一部分…… 也就是窗体中哪一部分“变脏”。然后.NET调用`OnPaint`，通知窗体触发一个`Paint`事件，并重绘脏的区域。

   实际上Invalidate()指出了窗体的某些部分可能“无效”，所以要重绘这一部分，以确保正常地显示。

3. **Update()方法为Invalidate请求提供最高优先级。**

   也许你还没有意识到，不过窗体确实一直都在接收消息。这个消息系统告诉窗体被覆盖并调用`OnPaint`，除此以外，这个消息系统中还有其他的各种消息需要发送。可以自己试试看：键入`override`，并滚动查看所有以“On”开头的方法，每个方法都是窗体需要响应的一个消息。`Update()`方法会把Invalidate消息移到这个消息列表的最前面。

   所以如果你自己调用Invalidate()，就是在告诉.NET：整个窗体或控件都无效，需要全部重绘。如果愿意，还可以传入你自己的剪裁矩形，这个矩形会进一步传递到Paint事件的PaintEventArgs。

4. **窗体的Refresh()方法就是Invalidate()加Update()。**

   窗体和控件提供了一个快捷方法。它们包含一个`Refresh()`方法，这个方法首先调用`Invalidate()`将整个客户区（窗体上图形出现的区域）置为无效，然后调用`Update()`确保将这个消息移到列表最前面。

## there are no Dumb Questions

**问：看起来还是在Paint或PhotoShop之类的程序中调整图片大小更好一些。为什么不能这样做呢？**

**答：**当然可以，如果应用中处理的图像都由你负责，而且它们的大小保持不变，确实可以这么做。但是通常情况并不是这样。大多数情况下，图像可能从其他来源得到，可能是在线获得，也可能是设计小组中的一个同事提供的。或者，有可能从一个只读来源读取图像，必须通过编程来调整它的大小。

**问：但是如果能在.NET之外调整图像大小，那样会更好一些，对不对？**

**答：**如果你确信不需要一个更大的版本，可能是这样。但是，如果你的程序可能需要在运行过程中以多种不同大小显示这个图像，就必须以某种方式在某个时刻调整它的大小。另外，如果显示图像时要比调整大小前更大，就会非常麻烦。缩小图像往往比放大容易得多。大多数情况下，最好能通过编程方式调整图像的大小，这样就不会受外部程序或某些约束（如只读文件）的限制。

**问：我看到CreateGraphics()得到了一个在窗体上绘图的Graphics对象，但是ResizeImage()方法中的FromImage()调用有什么作用？**

**答：**FromImage()为一个Bitmap对象获取Graphics对象。在窗体上调用CreateGraphics()会返回一个Graphics对象，用于在这个窗体上绘图，与此类似，FromImage()也会返回一个Graphics对象，可以用它在调用这个方法的Bitmap上绘图。

**问：这么说来，Graphics对象并不只是用于在窗体上绘图，对吗？**

**答：**确实如此，Graphics对象可以在任何对象上绘图，只要一个对象能提供Graphics对象，就可以用它在这个对象上绘图。Bitmap提供了一个Graphics对象，可以用来在一个不可见的图像上绘图，以便以后使用这个位图图像。除了窗体以外，你可能还经常在别处看到Graphics对象。将一个按钮拖到窗体上，然后在代码中键入这个按钮的名字，后面跟一个点号。查看弹出的智能提示窗口，其中将包括一个返回Graphics对象的CreateGraphics()方法。用这个对象绘制的所有内容都会显示在按钮上！对于Label、PictureBox、StatusStrip……以及几乎所有提供了Graphics对象的工具条控件来说都是如此。

**问：等一下，我以为using只能用于流。为什么对Graphics也要使用using呢？**

**答：**using关键字在处理流时很有用，不过只要类实现了IDisposable接口都可以使用using。一个类实现了IDisposable时，只要实例化这个类，删除实例时就应当调用其Dispose()方法。对于流来说，Dispose()方法可以确保已打开的任何文件都会关闭。Graphics，Pen和Brush对象都是可撤销的（实现了Disposable接口）。创建这些对象时，它们会占用少量内存和其他一些资源，而且不会立即交回。如果你只画一次，可能注意不到差别。但是大多数情况下，你的图形化代码会反复调用，例如Paint事件处理程序中的代码，对于一个特别忙的窗体，可能每秒要调用多次。正因如此，一定要调用Dispose()来撤销与图形有关的对象。为确保这一点，最容易的办法就是使用一个using语句，让.NET负责撤销的问题。用using创建的所有对象都会在using语句块结束时自动调用Dispose()方法。这就保证了程序运行很长时间时不会慢慢地占用越来越多的内存。

**问：如果我要创建一个新控件，应该使用UserControl，还是应该创建继承某个工具条控件的类？**

**答：**这要看你希望这个新控件做什么。如果要创建的控件与工具条中已有的某个控件非常相似，你会发现，最容易的办法就是继承那个控件创建派生类。但是大多数情况下，程序员用C#创建新控件时都会使用用户控件。用户控件有一个突出的优点，可以把工具条控件拖到用户控件上。它就像一个GroupBox或其他容器控件—可以把按钮或复选框拖到你的用户控件上，对这些工具条控件的处理与处理窗体上的控件完全相同。IDE的窗体设计工具是一个非常强大的工具，可以帮助你设计用户控件。稍后会更深入地了解用户控件。

**用户控件可以包含其他控件。IDE的窗体设计工具允许将工具条的控件拖到你的用户控件上。**

我注意到我的Zoomer控件在不停地闪烁。既然一直在说完全控制图形化，我相信肯定能在这方面做点什么！不过，为什么会产生这种闪烁呢？



**就算没有调整大小，把图像绘制到窗体上也是需要时间的。**

假设模拟系统中的每一个图像都已经调整好大小。要把所有这些蜜蜂和花以及蜂巢都画在窗体上，这也需要耗费一些时间。另外，现在是利用Graphics对象在窗体上绘图。所以如果你的眼睛能捕捉到渲染过程的最后阶段，就会发现有一点闪烁。问题就出在绘图工作太多，所以即使之前已经完成了图像的大小调整，仍然很可能出现闪烁。大多数业余计算机游戏存在问题也是因为这个原因……人眼能捕捉到渲染周期的最后阶段，就会认为在屏幕上有闪烁。

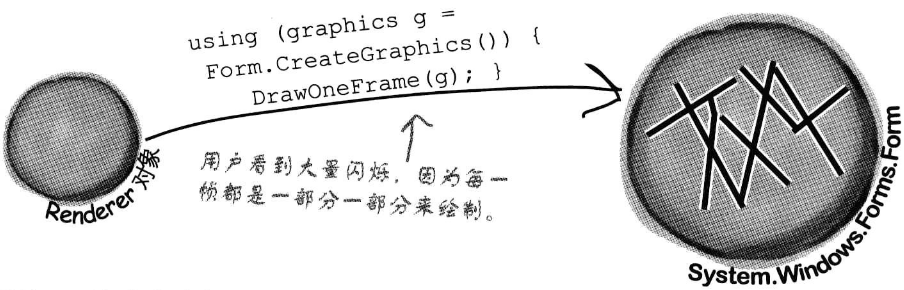
**BRAIN POWER**

怎么去除这种闪烁呢？如果在窗体上绘制太多图像会导致闪烁，而你必须绘制大量图像，你认为该如何避免闪烁？

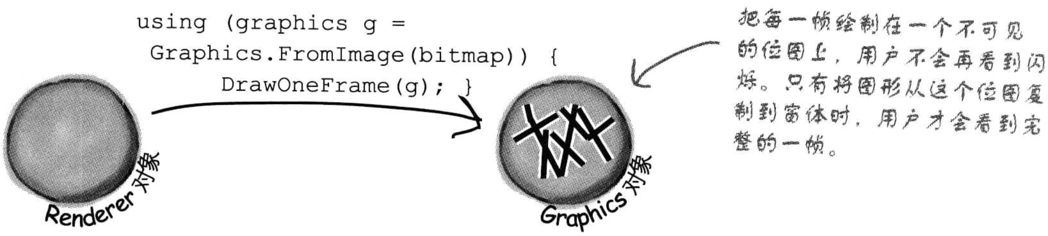
# 双缓冲使动画看起来更平滑

再来看前面构建的图像zoomer控件，任意移动trackbar上的滑块。注意到移动滑块时的闪烁了吗？这是因为Paint事件处理程序首先必须绘制白色矩形，然后每次trackbar移动一点点时就要绘制图像。你的眼睛会看到白色矩形和图像每秒切换多次，就会感觉有闪烁。这很让人恼火……使用一种称为双缓冲（double buffering）的技术完全能避免这个问题。这种技术是指，将每一帧或每个动画单元绘制到一个不可见的位图上（一个“缓冲区”），只在这一帧已经完全绘制时才显示下一帧。以下针对一个Bitmap来说明这是如何做到的：

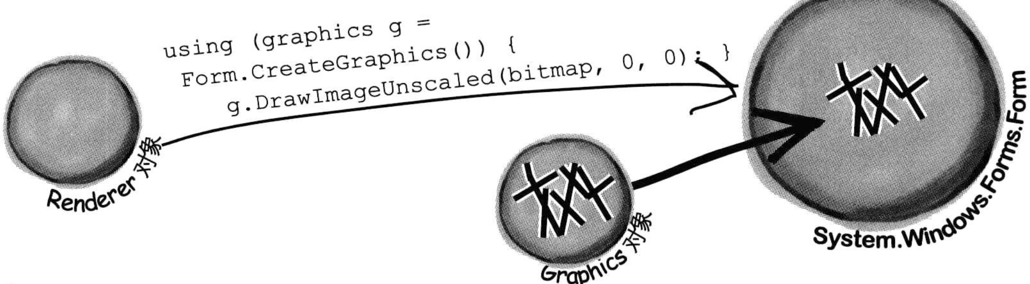
❶ 这是一个典型的程序，它使用窗体的Graphics对象在窗体上绘制一些图形。




❷ 要完成双缓冲，可以向程序增加一个Bitmap对象作为缓冲区。每次窗体或控件需要重绘时，并不是直接在窗体上绘制图形，而是在这个缓冲区上绘制。




❸ 既然一帧图像完全绘制到一个不可见的Bitmap对象上，可以使用DrawImageUnscaled()把这个Bitmap对象复制回窗体的Graphics。它会一次完全复制，这就能消除闪烁。



## 双缓冲已经内置到窗体和控件中

你可以使用Bitmap自行完成双缓冲，不过利用C#和.NET提供的内置支持，可以更容易地实现双缓冲。所要做的很简单，只需设置**DoubleBuffered**属性为true。可以在你的Zoomer用户控件上试试看，进入Properties窗口，设置DoubleBuffered为true，控件就会停止闪烁！下面再对BeeControl做同样的处理。尽管还不能修正所有图形化问题（稍后会做这个工作），但是现在确实大不相同了。

下面来修正蜂巢模拟系统中存在的图形化问题！

**使用Paint事件完成图形化处理时，只需改变一个属性就可以打开双缓冲绘制特性。**

## 全面检查蜂巢模拟系统

在下面的练习中，要对你的蜂巢模拟系统进行全面的检查修改。可能需要创建一个全新的工程，并使用“Add”→“Existing Item……”向这个工程增加现有的文件，得到当前模拟系统的一个备份（不要忘记修改这些文件的命名空间，要保证与你的新工程一致）。

你要做下面的工作：

1. **首先要删除BeeControl用户控件。**
蜂巢和花场上不会有任何控件。没有BeeControl,没有PictureBox，什么也没有。蜜蜂、花和蜂巢图片都使用GDI+图形库来绘制。所以在Solution Explorer中右键点击BeeControl.cs，并点击Delete—它们会从工程中去除，并永久删除。

2. **需要一个定时器处理蜜蜂扇动翅膀。**
蜜蜂扇动翅膀的速度比模拟系统的帧速率要慢得多，所以需要另一个更慢的定时器。这一点并不奇怪，因为BeeControl有它自己的定时器来实现蜜蜂翅膀的扇动。

3. **最重要的一步：修改Renderer。**
要完全抛开当前的Renderer，因为它的所有工作都依赖于控件完成。你并不需要那些字典，因为没有可查找的PictureBox或BeeControl。相反，Renderer要有两个重要的方法：DrawHive(g)用一个graphics对象绘制一个Hive窗体，DrawField(g)绘制一个Field窗体。

4. **最后要关联这个新Renderer。**
Hive和Field窗体都需要Paint事件处理程序。这两个事件处理程序将分别调用Renderer对象的DrawField(g)或DrawHive(g)方法。两个定时器（一个用于告诉模拟系统绘制下一帧，另一个用于扇动蜜蜂的翅膀）要调用两个窗体的Invalidate()方法自行重绘。此时它们的Paint事件处理程序就会显示这一帧。

**开始行动吧！** →

**Exercise**

现在来消除蜂巢模拟系统中存在的图形化问题。使用Graphics和双缓冲，使模拟系统更出色。

**1 修改主窗体的RunFrame()方法。**

要删除Renderer.Render()调用，并增加两个Invalidate()语句。

```
public void RunFrame(object sender, EventArgs e) {
    framesRun++;
    world.Go(random);
    end = DateTime.Now;
    TimeSpan frameDuration = end - start;
    start = end;
    UpdateStats(frameDuration);
    hiveForm.Invalidate();
    fieldForm.Invalidate();
}
```

需要删除renderer.Render()调用，因为这个方法已经不存在了。

只要让world保持最新，而且两个窗体都有Renderer对象的一个引用，要完成两个窗体的动画，只需调用它们的Invalidate()方法。窗体的Paint事件处理程序会负责余下的所有工作。

**2 向主窗体增加第二个定时器，完成蜜蜂的翅膀扇动。**

把一个新定时器拖到主窗体上，设置其Interval等于150ms，并将Enabled设置为true。然后双击这个定时器，增加以下事件处理程序：

```
private void timer2_Tick(object sender, EventArgs e) {
    renderer.AnimateBees();
}
```

然后向Renderer增加以下AnimateBees()方法，使蜜蜂的翅膀扇动：

```
private int cell = 0;
private int frame = 0;
public void AnimateBees() {
    frame++;
    if (frame >= 6)
    frame = 0;
    switch (frame) {
        case 0: cell = 0; break;
        case 1: cell = 1; break;
        case 2: cell = 2; break;
        case 3: cell = 3; break;
        case 4: cell = 2; break;
        case 5: cell = 1; break;
        default: cell = 0; break;
    }
    hiveForm.Invalidate();
    fieldForm.Invalidate();
}
```

这里的关键是设置一个名为cell的字段，Renderer绘制蜜蜂时会用到这个字段。要保证蜂巢窗体中绘制的是BeeAnimationLarge[cell]，花场窗体中则要绘制BeeAnimationSmall[cell]。定时器会不停地调用AnimateBees()方法，使cell字段一直在改变，这就使得蜜蜂在一直扇动翅膀。

**如果蜜蜂飞到错误的地方，要确保你的位置是正确的！使用本章前面MouseClick事件的有关技巧得出正确的坐标。**

**3 蜂巢窗体和花场窗体都需要一个公共的Renderer属性。**

为蜂巢窗体和花场窗体增加一个公共字段：

```
public Renderer Renderer { get; set; }
```

两个窗体都要增加这行代码

不要忘记增加这些访问修饰符！

要让这起作用，需要修改Renderer的声明增加public修饰符，变成：`public class Renderer`。还需要对`World`、`Hive`、`Bee`和`Flower`类以及`BeeState enum`做同样的处理—为它们的声明分别增加public访问修饰符。查看附录“其他”中的第2项了解这是为什么！

有两个地方创建了新的Renderer：一处是在open按钮中（在`renderer.Reset()`调用下面），另一处是在`ResetSimulator()`方法中。删除所有`renderer.Reset()`调用，更新Renderer的构造函数来设置各个窗体的Renderer属性：

```
    hiveForm.Renderer = this;
    fieldForm.Renderer = this;
```

Reset()方法所做的就是从窗体删除控件，但现在已经没有要删除的控件。

**4 设置蜂巢和花场窗体完成双缓冲动画。**

从蜂巢窗体的构造函数中删除设置背景图像的代码。然后从两个窗体中删除所有控件，并设置这两个窗体的**`DoubleBuffered`**属性为**`true`**。最后，向这两个窗体分别增加一个`Paint`事件处理程序。以下是蜂巢窗体的`Paint`事件处理程序，花场窗体的`Paint`事件处理程序也一样，只不过要调用`Renderer.PaintField()`而不是`Renderer.PaintHive()`：

```
private void HiveForm_Paint(object sender, PaintEventArgs e) {
    Renderer.PaintHive(e.Graphics);
}
```

要保证已经打开了双缓冲，否则你的窗体会闪烁！

**5 全面修改Renderer，删除基于控件的代码，而增加图形化代码。**

修正Renderer需要做以下工作：

- ★ 删除两个字典，因为已经没有控件了。而且既然如此，不再需要`BeeControl`，也不需要`Render()`，`DrawBees()`或`DrawFlowers()`方法。
- ★ 增加一些`Bitmap`字段：`HiveInside`、`HiveOutside`和`Flower`，用来保存图像。然后创建两个`Bitmap[]`数组，名为`BeeAnimationLarge`和`BeeAnimationSmall`。各数组分别包含4个蜜蜂图片，大图片大小为40×40，小图片为20×20。创建一个`InitializeImages()`方法，调整这些资源的大小，并把它们保存到相应字段中，从Renderer类的构造函数调用这个方法。
- ★ 增加`PaintHive()`方法，它取一个`Graphics`对象作为参数，并用这个对象绘制蜂巢窗体。首先画一个天蓝色的矩形，然后使用`DrawImageUnscaled()`绘制蜂巢内部图片，再使用`DrawImageUnscaled()`绘制蜂巢里的各个蜜蜂。
- ★ 最后，增加`PaintField()`方法。它要在窗体的上半部分绘制一个天蓝色的矩形，下半部分绘制一个绿色的矩形。你会发现有两个窗体属性对此很有帮助，`ClientSize`和`ClientRectangle`属性能告诉你绘制区有多大，所以可以使用`ClientSize.Height/2`得出其高度的一半。然后使用`FillEllipse()`在天空上画一个黄色的太阳，使用`DrawLine()`画一个粗线作为挂蜂巢的杆，并使用`DrawImageUnscaled()`绘制蜂巢外部图片。然后把每朵花画在这个窗体上。最后，画各个蜜蜂（使用小蜜蜂图片）。之所以最后才画蜜蜂，是为了让它们出现在花的前面。
- ★ 绘制蜜蜂时，记住`AnimateBees()`要设置`Cell`字段。

**Exercise Solution**

现在来消除蜂巢模拟系统中存在的图形化问题。使用Graphics和双缓冲，使模拟系统更出色。

这是完整的Renderer类，包括前面已经提供的AnimateBees()方法。要保证已经对3个窗体（主窗体、蜂巢窗体和花场窗体）完成了所有修改，特别是蜂巢和花场窗体中的Paint事件处理程序。这些事件处理程序调用Renderer的PaintHive()和PaintField()方法来完成所有动画。

```
using System.Drawing;

public class Renderer {
    private World world;
    private HiveForm hiveForm;
    private FieldForm fieldForm;

    public Renderer(World TheWorld, HiveForm hiveForm, FieldForm fieldForm) {
        this.world = TheWorld;
        this.hiveForm = hiveForm;
        this.fieldForm = fieldForm;
        fieldForm.Renderer = this;
        hiveForm.Renderer = this;
        InitializeImages();
    }
```

* 不要忘记修改Renderer.cs中的类声明，要由class Renderer改为public class Renderer，然后对World、Hive、Flower和Bee做同样的处理；否则，你会收到一个有关字段和类型可访问性的编译错误。可以翻到附录“其他”中的第2项了解为什么需要这么做。

```
    public static Bitmap ResizeImage(Image ImageToResize, int Width, int Height) {
        Bitmap bitmap = new Bitmap(Width, Height);
        using (Graphics graphics = Graphics.FromImage(bitmap)) {
            graphics.DrawImage(ImageToResize, 0, 0, Width, Height);
        }
        return bitmap;
    }
```

InitializeImages()方法调整所有图像资源的大小，并把它们存储在Renderer对象的Bitmap字段中。这样一来，PaintHive()和PaintForm()方法可以使用窗体Graphics对象的DrawImageUnscaled()方法无需放缩地绘制图像。

```
    Bitmap HiveInside;
    Bitmap HiveOutside;
    Bitmap Flower;
    Bitmap[] BeeAnimationSmall;
    Bitmap[] BeeAnimationLarge;
    private void InitializeImages() {
        HiveOutside = ResizeImage(Properties.Resources.Hive__outside_, 85, 100);
        Flower = ResizeImage(Properties.Resources.Flower, 75, 75);
        HiveInside = ResizeImage(Properties.Resources.Hive__inside_,
            hiveForm.ClientRectangle.Width, hiveForm.ClientRectangle.Height);
        BeeAnimationLarge = new Bitmap[4];
        BeeAnimationLarge[0] = ResizeImage(Properties.Resources.Bee_animation_1, 40, 40);
        BeeAnimationLarge[1] = ResizeImage(Properties.Resources.Bee_animation_2, 40, 40);
        BeeAnimationLarge[2] = ResizeImage(Properties.Resources.Bee_animation_3, 40, 40);
        BeeAnimationLarge[3] = ResizeImage(Properties.Resources.Bee_animation_4, 40, 40);
        BeeAnimationSmall = new Bitmap[4];
        BeeAnimationSmall[0] = ResizeImage(Properties.Resources.Bee_animation_1, 20, 20);
        BeeAnimationSmall[1] = ResizeImage(Properties.Resources.Bee_animation_2, 20, 20);
        BeeAnimationSmall[2] = ResizeImage(Properties.Resources.Bee_animation_3, 20, 20);
        BeeAnimationSmall[3] = ResizeImage(Properties.Resources.Bee_animation_4, 20, 20);
    }
```

```
    public void PaintHive(Graphics g) {
        g.FillRectangle(Brushes.SkyBlue, hiveForm.ClientRectangle);
        g.DrawImageUnscaled(HiveInside, 0, 0);
        foreach (Bee bee in world.Bees) {
            if (bee.InsideHive)
                g.DrawImageUnscaled(BeeAnimationLarge[cell],
                                    bee.Location.X, bee.Location.Y);
        }
    }

    public void PaintField(Graphics g) {
        using (Pen brownPen = new Pen(Color.Brown, 6.0F)) {
            g.FillRectangle(Brushes.SkyBlue, 0, 0,
                     fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
            g.FillEllipse(Brushes.Yellow, new RectangleF(50, 15, 70, 70));
            g.FillRectangle(Brushes.Green, 0, fieldForm.ClientSize.Height / 2,
                     fieldForm.ClientSize.Width, fieldForm.ClientSize.Height / 2);
            g.DrawLine(brownPen, new Point(593, 0), new Point(593, 30));
            g.DrawImageUnscaled(HiveOutside, 550, 20);
            foreach (Flower flower in world.Flowers) {
                g.DrawImageUnscaled(Flower, flower.Location.X, flower.Location.Y);
            }
            foreach (Bee bee in world.Bees) {
                if (!bee.InsideHive)
                    g.DrawImageUnscaled(BeeAnimationSmall[cell],
                                        bee.Location.X, bee.Location.Y);
            }
        }
    }

    private int cell = 0;
    private int frame = 0;
    public void AnimateBees() {
        frame++;
        if (frame >= 6)
        frame = 0;
        switch (frame) {
            case 0: cell = 0; break;
            case 1: cell = 1; break;
            case 2: cell = 2; break;
            case 3: cell = 3; break;
            case 4: cell = 2; break;
            case 5: cell = 1; break;
            default: cell = 0; break;
        }
        hiveForm.Invalidate();
        fieldForm.Invalidate();
    }

}
```

窗体的ClientSize属性是一个Size，可以告诉你它的绘制区有多大。

PaintField()方法查看这个世界里的蜜蜂和花，使用它们的位置来绘制一个花场。首先，绘制天空和地面，然后画出太阳。再画出蜂巢。接下来，画出花和蜜蜂。要按正确的顺序来完成绘制，这一点很重要，如果先画蜜蜂再画花，看起来蜜蜂就会像是在花的后面飞。

这正是练习中提供的AnimateBees()方法。它使用Frame字段循环处理动画，首先显示cell 0，然后是cell 1，接下来2，然后3，再回到2，再到1。这样蜜蜂的动画就是平滑的。

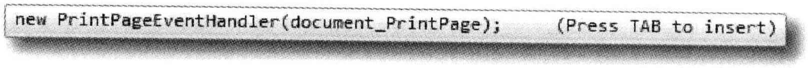
# 使用Graphics对象和事件处理程序完成打印

前面在窗体上绘图时使用了Graphics的方法，打印时也要使用同样的方法。.NET的打印对象都在System.Drawing.Printing中，利用这些对象，可以很容易地为应用增加打印和打印预览功能。只需创建一个PrintDocument对象。它有一个事件PrintPage，可以像定时器的Tick事件一样使用这个PrintPage事件。然后调用PrintDocument对象的Print()方法，它会打印文档。要记住，利用IDE增加事件处理程序非常容易。步骤如下：

**动手打印**

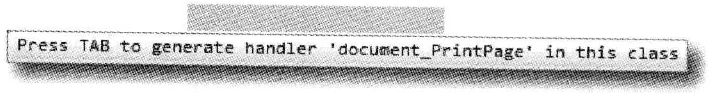
1. **启动一个新的Windows应用**，向窗体增加一个按钮。打开窗体代码，在最上面增加一行 **using System.Drawing.Printing;**。双击按钮，增加事件处理程序。观察键入+=时会发生什么：

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage +=
```




2. 按下Tab键，IDE会自动填完这一行余下的代码。这类似于第11章中增加事件处理程序：

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
```




3. 键入Tab后，IDE生成一个事件处理方法，并增加到窗体。

```
void document_PrintPage(object sender, PrintPageEventArgs e) {
    throw new NotImplementedException();
}
```

现在可以在这里增加任何图形化代码，只要替换这行throw代码，使用e.Graphics来完成所有绘图。稍后将介绍如何实现……

PrintPageEventArgs参数e有一个Graphics属性。只需用e.Graphics对象的绘制方法调用替换throw语句。

4. 下面完成button1 _ Click事件处理程序，调用**document.Print()**。调用这个方法时，PrintDocument对象创建一个Graphics对象，然后触发一个PrintPage事件，以这个Graphics对象作为参数。这个事件处理程序画到Graphics对象上的所有内容都将发送到打印机。

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    document.Print();
}
```

# PrintDocument处理打印对话框和打印预览窗口对象

要增加一个打印预览窗口或打印对话框，这与增加一个打开或保存对话框非常相似。只需创建一个PrintDialog或PrintPreviewDialog对象，设置其Document属性为所要打印的Document对象，然后调用这个对话框的Show()方法。对话框会负责把文档发送到打印机，不需要调用它的Print()方法。下面把这个功能增加到第1步创建的按钮上：

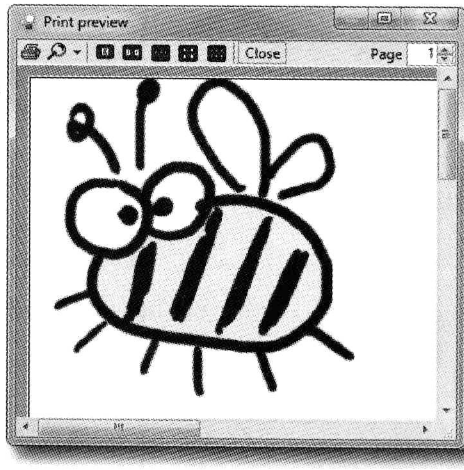
一旦有了PrintDocument和完成打印的事件处理程序，可以弹出一个打印预览窗口，为此只需创建一个新的PrintPreviewDialog对象。

5

```
private void button1_Click(object sender, EventArgs e) {
    PrintDocument document = new PrintDocument();
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    PrintPreviewDialog preview = new PrintPreviewDialog();
    preview.Document = document;
    preview.ShowDialog(this);
}

void document_PrintPage(object sender,
                        PrintPageEventArgs e) {
    DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
}
```

重用前几页编写的DrawBee()方法。




## 使用e.HasMorePages打印多页文档

如果需要打印多页，所要做的就是让PrintPage事件处理程序设置e.HasMorePages为true。这会告诉Document，还有下一页需要打印。它会反复调用这个事件处理程序，只要事件处理程序一直将e.HasMorePages设置为true，对文档中的每一页都会调用一次事件处理程序。下面修改Document的事件处理程序来打印两页：

6

```
bool firstPage = true;
void document_PrintPage(object sender, PrintPageEventArgs e) {
   DrawBee(e.Graphics, new Rectangle(0, 0, 300, 300));
   using (Font font = new Font("Arial", 36, FontStyle.Bold)) {
       if (firstPage) {
           e.Graphics.DrawString("First page", Font, Brushes.Black, 0, 0);
           e.HasMorePages = true;
           firstPage = false;
       } else {
           e.Graphics.DrawString("Second page", Font, Brushes.Black, 0, 0);
           firstPage = true;
       }
   }
}
```

如果设置e.HasMorePages为true，Document对象会再次调用事件处理程序来打印下一页。

现在再运行你的程序，确保它在打印预览中显示两页。




在模拟系统中为Print按钮编写代码，使它弹出一个打印预览窗口显示蜜蜂统计信息和蜂巢及花场的图片。

**1 让按钮弹出一个打印预览窗口。**

为按钮的点击事件增加一个事件处理程序，它会暂停模拟系统，弹出打印预览对话框，然后继续模拟系统的运行。（如果点击按钮时模拟系统本来就暂停，那么显示了预览窗口之后还应当是暂停的）。

**2 创建文档的PrintPage事件处理程序。**

创建一个与下一页中类似的页面。可以利用下面的代码作为基础：

```
private void document_PrintPage(object sender, PrintPageEventArgs e) {
    Graphics g = e.Graphics;
    Size stringSize;
    using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
        stringSize = Size.Ceiling(
              g.MeasureString("Bee Simulator", arial24bold));
        g.FillEllipse(Brushes.Gray,
            new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
            stringSize.Width + 30, stringSize.Height + 30));
        g.FillEllipse(Brushes.Black,
            new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
            stringSize.Width + 30, stringSize.Height + 30));
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
        g.DrawString("Bee Simulator", arial24bold,
            Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
    }
    int tableX = e.MarginBounds.X + (int)stringSize.Width + 50;
    int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;
    int firstColumnX = tableX + 2;
    int secondColumnX = tableX + (tableWidth / 2) + 5;
    int tableY = e.MarginBounds.Y;
    // 你的任务：填入这个方法的其余代码，让它打印这个页面
```

使用MeasureString()方法创建一个包含文本的椭圆，这会返回一个Size，包含串的大小。我们将椭圆和文本绘制两次，提供一种阴影效果。

需要这些代码来建立表格。

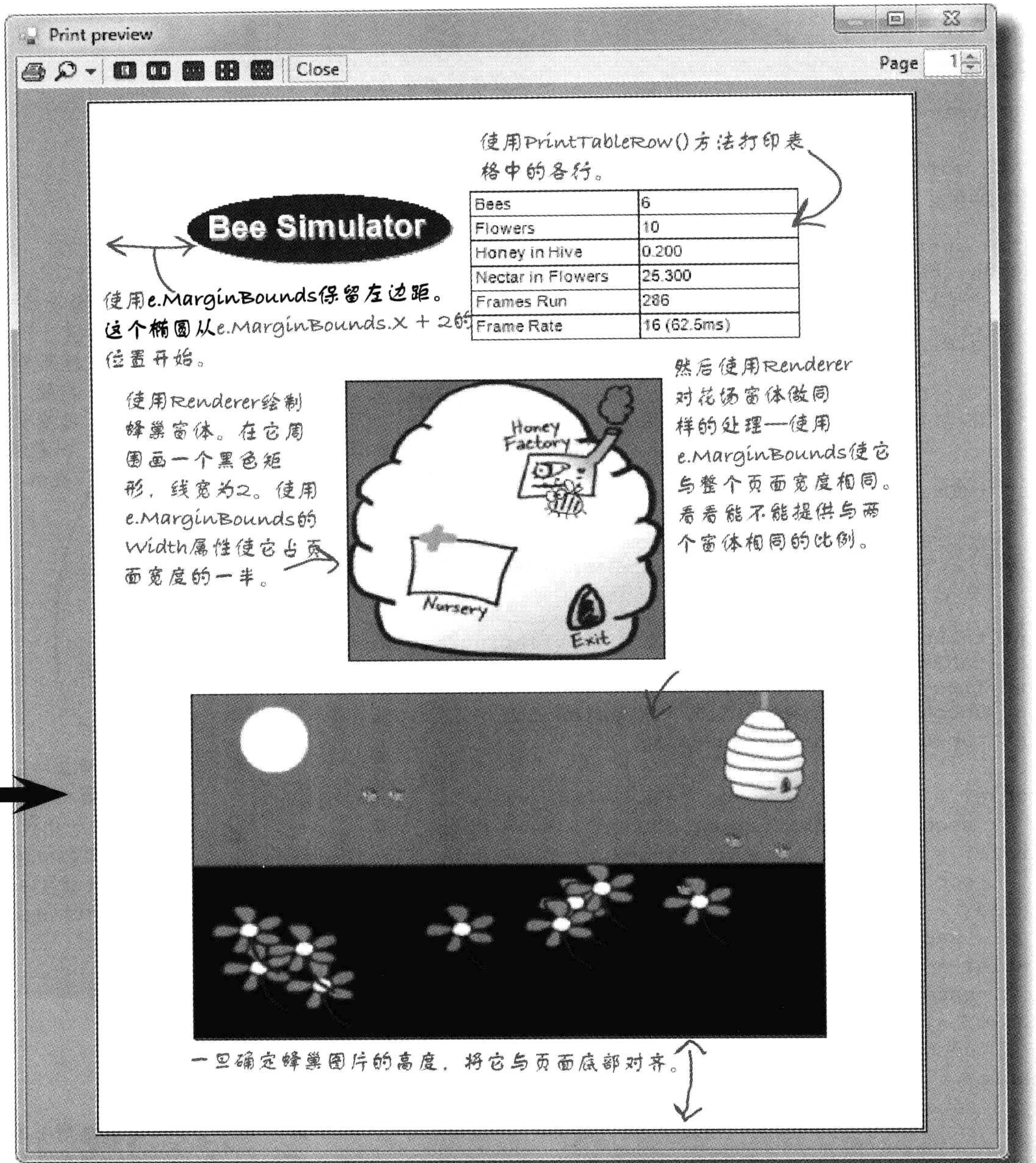
**3 这个PrintTableRow()方法很方便。**

在页面最上面创建蜜蜂的统计信息表时，你会发现这个方法很有用。

```
private int PrintTableRow(Graphics printGraphics, int tableX,
                 int tableWidth, int firstColumnX, int secondColumnX,
                 int tableY, string firstColumn, string secondColumn) {
  Font arial12 = new Font("Arial", 12);
  Size stringSize = Size.Ceiling(printGraphics.MeasureString(firstColumn, arial12));
  tableY += 2;
  printGraphics.DrawString(firstColumn, arial12, Brushes.Black,
                 firstColumnX, tableY);
  printGraphics.DrawString(secondColumn, arial12, Brushes.Black,
                 secondColumnX, tableY);
  tableY += (int)stringSize.Height + 2;
  printGraphics.DrawLine(Pens.Black, tableX, tableY, tableX + tableWidth, tableY);
  arial12.Dispose();
  return tableY;
}
```

每次调用PrintTableRow()时，它会把所打印那一行的行高增加到tableY，并返回这个新值。

**仔细看看我们对打印页面做出的提示。这稍微有些复杂，要多花点时间考虑！**




**提示**：要得出每个窗体的高度，可以先得到其高度除以宽度的比例，再乘以最终的宽度。可以将从页面的下边距减去花场窗体的高度来确定它顶部的位置：(e.MarginBounds.Y + e.MarginBounds.Height - fieldHeight)。




在模拟系统中为Print按钮编写代码，使它弹出一个打印预览窗口显示蜜蜂统计信息和蜂巢及花场的图片。

这是Document的PrintPage事件处理程序。要放在窗体中。

```
using System.Drawing.Printing;

private void document_PrintPage(object sender, PrintPageEventArgs e) {
   Graphics g = e.Graphics;

   Size stringSize;
   using (Font arial24bold = new Font("Arial", 24, FontStyle.Bold)) {
       stringSize = Size.Ceiling(
            g.MeasureString("Bee Simulator", arial24bold));
       g.FillEllipse(Brushes.Gray,
           new Rectangle(e.MarginBounds.X + 2, e.MarginBounds.Y + 2,
           stringSize.Width + 30, stringSize.Height + 30));
       g.FillEllipse(Brushes.Black,
           new Rectangle(e.MarginBounds.X, e.MarginBounds.Y,
           stringSize.Width + 30, stringSize.Height + 30));
       g.DrawString("Bee Simulator", arial24bold,
           Brushes.Gray, e.MarginBounds.X + 17, e.MarginBounds.Y + 17);
       g.DrawString("Bee Simulator", arial24bold,
           Brushes.White, e.MarginBounds.X + 15, e.MarginBounds.Y + 15);
   }

   int tableX = e.MarginBounds.X + (int)stringSize.Width + 50;
   int tableWidth = e.MarginBounds.X + e.MarginBounds.Width - tableX - 20;
   int firstColumnX = tableX + 2;
   int secondColumnX = tableX + (tableWidth / 2) + 5;
   int tableY = e.MarginBounds.Y;

   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Bees", Bees.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Flowers", Flowers.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
         secondColumnX, tableY, "Honey in Hive", HoneyInHive.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Nectar in Flowers", NectarInFlowers.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Frames Run", FramesRun.Text);
   tableY = PrintTableRow(g, tableX, tableWidth, firstColumnX,
        secondColumnX, tableY, "Frame Rate", FrameRate.Text);

   g.DrawRectangle(Pens.Black, tableX, e.MarginBounds.Y,
        tableWidth, tableY - e.MarginBounds.Y);
   g.DrawLine(Pens.Black, secondColumnX, e.MarginBounds.Y,
        secondColumnX, tableY);
```

前面已经提供了这一部分代码。这里画出了椭圆标题，并建立一些变量用于绘制蜜蜂统计信息表格。

能不能看出PrintTableRow()方法是怎样做的？只需对每一行调用一次这个方法，它会把你希望打印的文本分两列打印。关键在于它会为下一行返回新的tableY值。

不要忘记在表格周围画一个矩形，另外还要在列之间画线。

需要一个黑色画笔（2个像素宽）在屏幕周围画线。

位图的大小应该与窗体的绘制区相同，因此可以用到ClientSize。

由于画笔和两个位图需要撤销，所以它们都放在一个大的using块中。

```
using (Pen blackPen = new Pen(Brushes.Black, 2))
using (Bitmap hiveBitmap = new Bitmap(hiveForm.ClientSize.Width,
                                      hiveForm.ClientSize.Height))
using (Bitmap fieldBitmap = new Bitmap(fieldForm.ClientSize.Width,
                                       fieldForm.ClientSize.Height))
{
    using (Graphics hiveGraphics = Graphics.FromImage(hiveBitmap))
    {
        renderer.PaintHive(hiveGraphics);
    }

    int hiveWidth = e.MarginBounds.Width / 2;
    float ratio = (float)hiveBitmap.Height / (float)hiveBitmap.Width;
    int hiveHeight = (int)(hiveWidth * ratio);
    int hiveX = e.MarginBounds.X + (e.MarginBounds.Width - hiveWidth) / 2;
    int hiveY = e.MarginBounds.Height / 3;
    g.DrawImage(hiveBitmap, hiveX, hiveY, hiveWidth, hiveHeight);
    g.DrawRectangle(blackPen, hiveX, hiveY, hiveWidth, hiveHeight);

    using (Graphics fieldGraphics = Graphics.FromImage(fieldBitmap))
    {
        renderer.PaintField(fieldGraphics);
    }
    int fieldWidth = e.MarginBounds.Width;
    ratio = (float)fieldBitmap.Height / (float)fieldBitmap.Width;
    int fieldHeight = (int)(fieldWidth * ratio);
    int fieldX = e.MarginBounds.X;
    int fieldY = e.MarginBounds.Y + e.MarginBounds.Height - fieldHeight;
    g.DrawImage(fieldBitmap, fieldX, fieldY, fieldWidth, fieldHeight);
    g.DrawRectangle(blackPen, fieldX, fieldY, fieldWidth, fieldHeight);
}
}
```

PaintHive()方法需要一个Graphics对象完成绘制，所以这个代码创建一个空的Bitmap对象，并传入PaintHive()。

e.MarginBounds.Width为页面可打印区的宽度。显示花场图片时也设置为这个宽度。

在这里使用窗体的高度-宽度比例来计算屏幕的高度。

```
private void printToolStripButton1_Click(object sender, EventArgs e) {
    bool stoppedTimer = false;
    if (timer1.Enabled) {
        timer1.Stop();
        stoppedTimer = true;
    }
    PrintPreviewDialog preview = new PrintPreviewDialog();
    PrintDocument document = new PrintDocument();
    preview.Document = document;
    document.PrintPage += new PrintPageEventHandler(document_PrintPage);
    preview.ShowDialog(this);
    if (stoppedTimer)
        timer1.Start();
}
```

这是Print按钮的代码。它会暂停模拟系统（如果正在运行），创建一个PrintDocument，把它关联到PrintPage事件处理程序，显示对话框，然后重新启动模拟系统。

# 还有一些工作可以做……

你已经构建了一个很不错的小型模拟系统，不要停下脚步，还有很多工作你可以自己来完成。下面给出一些想法，看看你能不能实现其中一部分。

## 增加一个控制面板

把World和Hive类中的常量变为属性。然后增加一个新窗体，其中包含一个控制面板，提供一些滑动条来控制这些属性。

## 增加敌人

增加攻击蜂巢的敌人。花越多，就有越多的敌人被吸引到蜂巢。再增加带刺巡逻蜂（Sting Patrol）保卫蜂巢，阻止敌人入侵，并增加蜂巢维护蜂（Hive Maintenance）保护并修补蜂巢。这些蜜蜂需要更多蜂蜜。

一个好的模拟系统应该有很多有利与不利因素的权衡，应该能够让用户决定哪些权衡来影响蜂巢的发展。

## 增加蜂巢升级

如果蜂巢得到足够多的蜂蜜，它可以更大一些。更大的蜂巢可以容纳更多的蜜蜂，不过也会消耗更多的蜂蜜，并吸引更多的敌人。如果敌人带来过大的破坏，蜂巢就会再次变小。

## 增加一个能产卵的蜂王

这些卵需要保育蜂（Baby Bee Care）来照顾。蜂巢里的蜂蜜越多，蜂王就能产更多的卵，这就需要更多的工蜂来照顾它们，相应地会消耗更多的蜂蜜。

## 增加动画

让Hive窗体的背景实现动画，使太阳缓慢地移过天空。晚上时变暗，并画上星星和月亮。再增加一些视角效果，如果蜜蜂在花场上，离蜂巢越远的蜜蜂就越小。

## 发挥你自己的想像力！

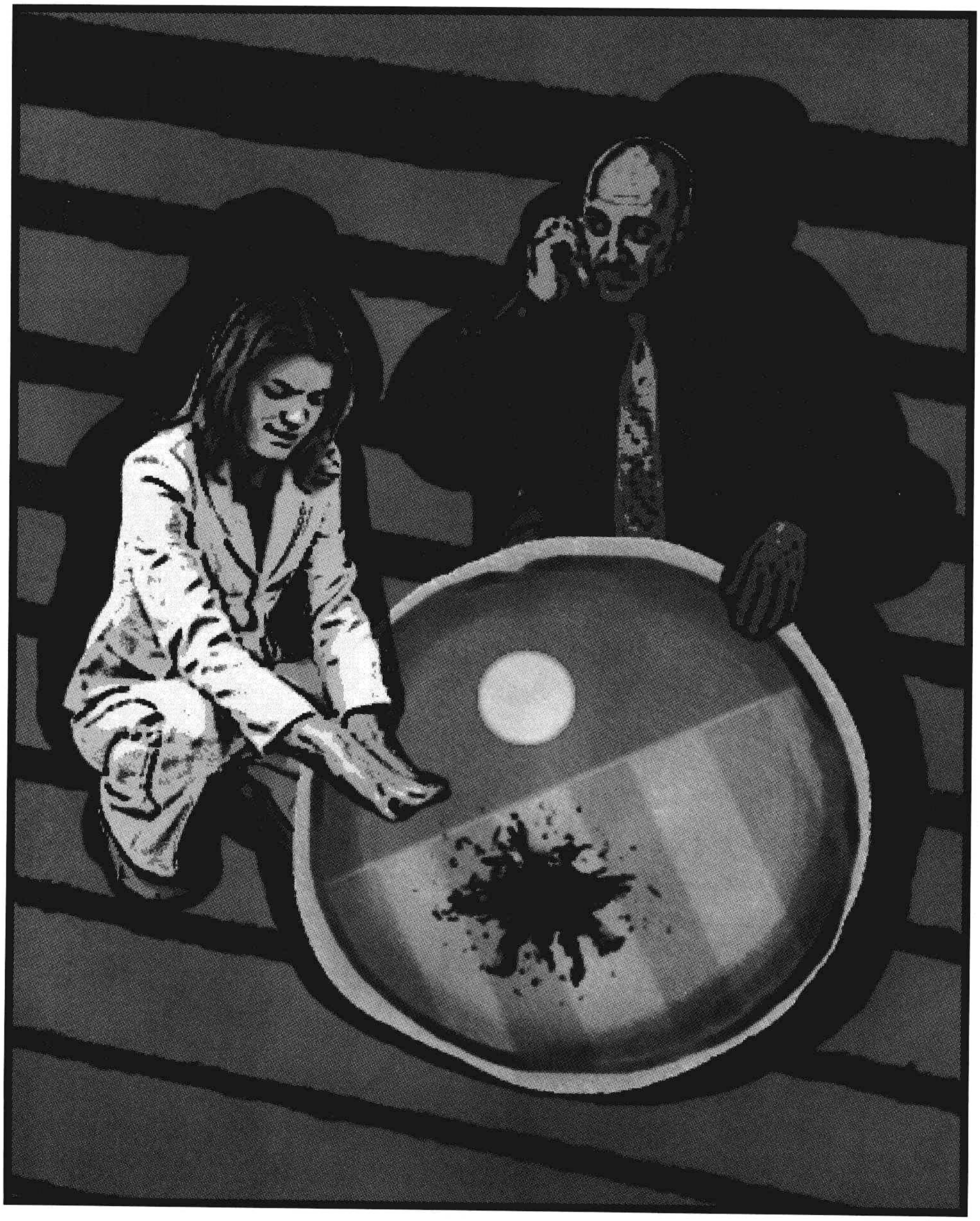
想想看，还能采用什么方式让这个模拟系统更有意思、更吸引人。

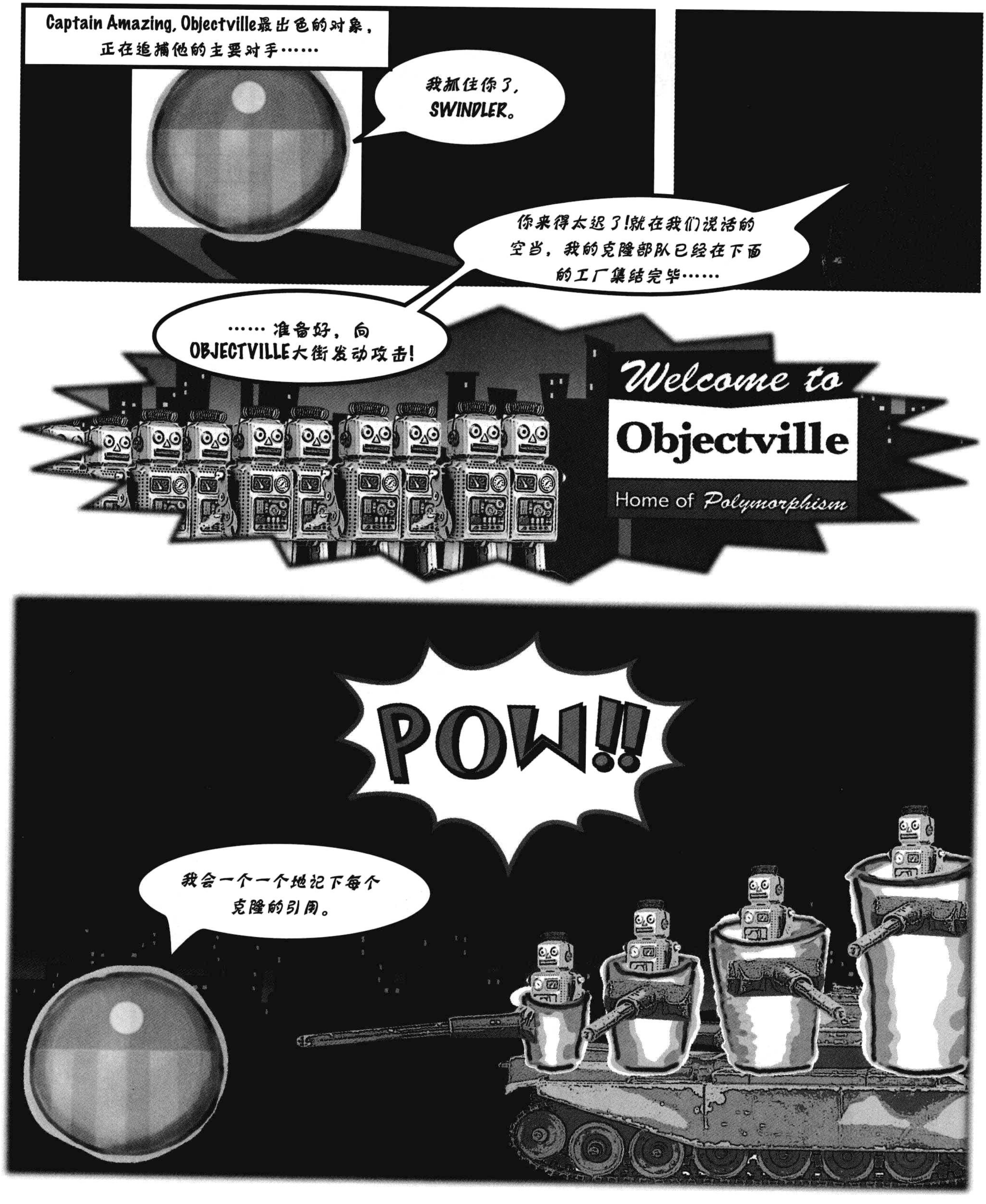
**你能修改这个模拟系统，提供一个更酷的版本吗？可以把你的工程源代码上传到Head First C# 论坛（www.headfirstlabs.com/books/hfcsharp/），展示你的高超水平。**

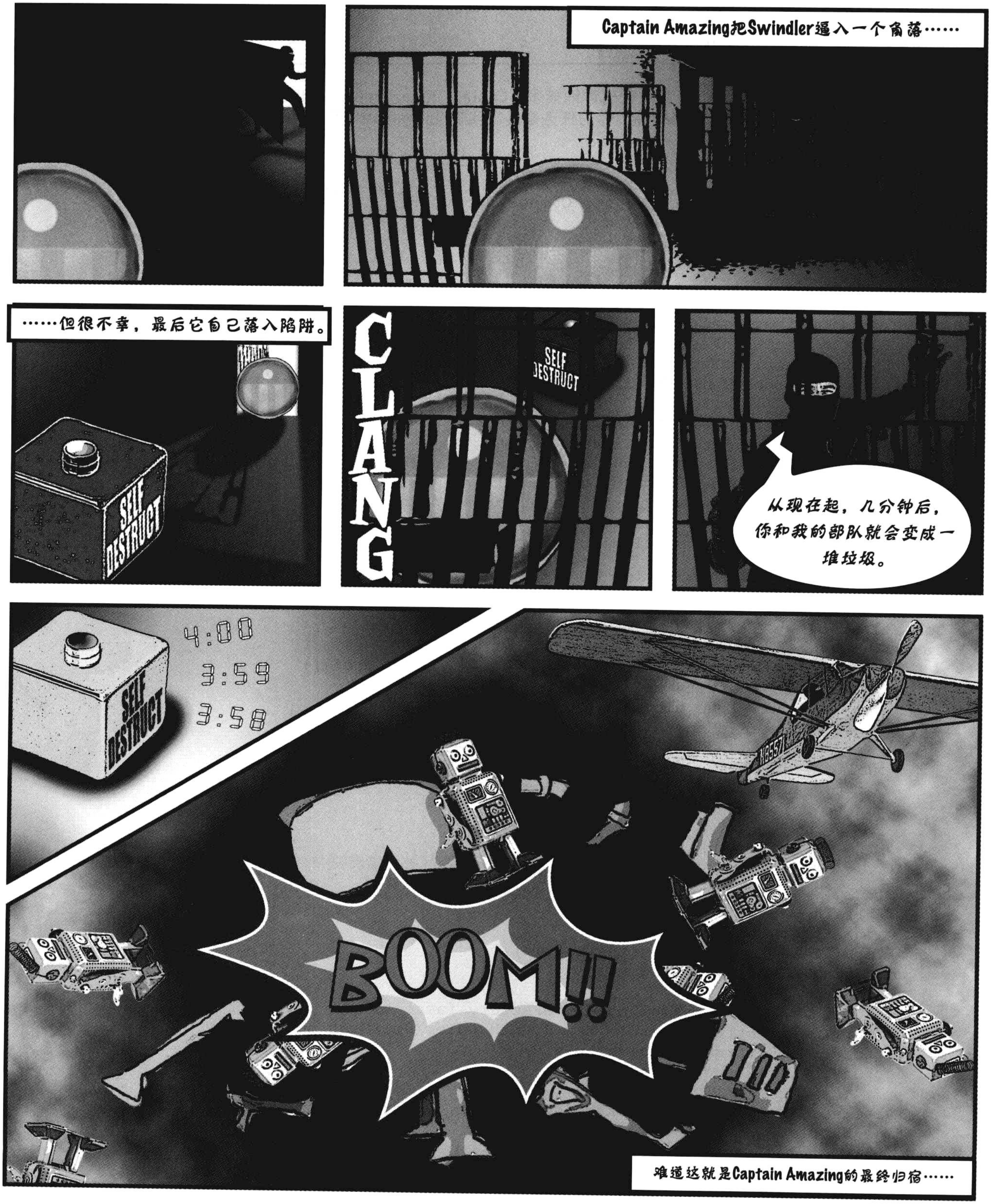
# CAPTAIN AMAZING

## 对象之死

| Head First Labs | |
|---|---|
| $2.98 | 第14章 |




Captain Amazing, Objectville最出色的对象，正在追捕他的主要对手……
我抓住你了，SWINDLER。
你来得太迟了!就在我们说话的空当，我的克隆部队已经在下面的工厂集结完毕……
…… 准备好，向OBJECTVILLE大街发动攻击!
Welcome to
Objectville
Home of Polymorphism
POW!!
我会一个一个地记下每个克隆的引用。



Captain Amazing把Swindler逼入一个角落……
……但很不幸，最后它自己落入陷阱。
SELF DESTRUCT
CLANG
SELF DESTRUCT
从现在起，几分钟后，你和我的部队就会变成一堆垃圾。
SELF DESTRUCT
4:00
3:59
3:58
BOOM!!
难道这就是Captain Amazing的最终归宿……

## Sharpen your pencil

下面的代码详细描述了Captain Amazing和Swindler（先不考虑他的克隆部队）之间的战斗。你的任务是画出实例化FinalBattle类时内存中有什么。

可以认为Clones是使用一个集合初始化方法设置的。

从这里开始，首先工厂对象中新增了内容。

画出实例化SwindlersEscapePlane对象时会发生什么。

画出FinalBattle构造函数运行1秒之后堆的状态图。

```
class FinalBattle {
  public CloneFactory Factory = new CloneFactory();
  public List<Clone> Clones = new List<Clone>() { ... };
  public SwindlersEscapePlane escapePlane;

  public FinalBattle() {
    Villain swindler = new Villain(this);
    using (Superhero captainAmazing = new Superhero()) {
      Factory.PeopleInFactory.Add(captainAmazing);
      Factory.PeopleInFactory.Add(swindler); ❶
      captainAmazing.Think("I'll take down each clone's reference,
                            one by one");
      captainAmazing.IdentifyTheClones(Clones);
      captainAmazing.RemoveTheClones(Clones);
      swindler.Think("A few minutes from now, you AND my army will be garbage");
      swindler.Think("(collected, that is!)");
      escapePlane = new SwindlersEscapePlane(swindler); ❷
      swindler.TrapCaptainAmazing(Factory);
      MessageBox.Show("The Swindler escaped");
    }
  }
} ❸
[Serializable]
class Superhero : IDisposable {
      private List<Clone> clonesToRemove = new List<Clone>();
      public void IdentifyTheClones(List<Clone> clones) {
          foreach (Clone clone in clones)
              clonesToRemove.Add(clone);
      }
      public void RemoveTheClones(List<Clone> clones) {
          foreach (Clone clone in clonesToRemove)
              clones.Remove(clone);
       ...
      }
      ...
}
class Villain {
  private FinalBattle finalBattle;
  public Villain(FinalBattle finalBattle) {
    this.finalBattle = finalBattle;
  }
  public void TrapCaptainAmazing(CloneFactory factory) {
    factory.SelfDestruct.Tick += new EventHandler(SelfDestruct_Tick);
    factory.SelfDestruct.Interval = 600;
    factory.SelfDestruct.Start();
  }
  private void SelfDestruct_Tick(object sender, EventArgs e) {
    finalBattle.Factory = null;
  }
}
```

还有其他一些代码在此没有显示（包括实现IDisposable的Dispose()方法），不过回答这个问题不需要这些代码。

```
class SwindlersEscapePlane {
  public Villain PilotsSeat;
  public SwindlersEscapePlane(Villain escapee) {
    PilotsSeat = escapee;
  }
}

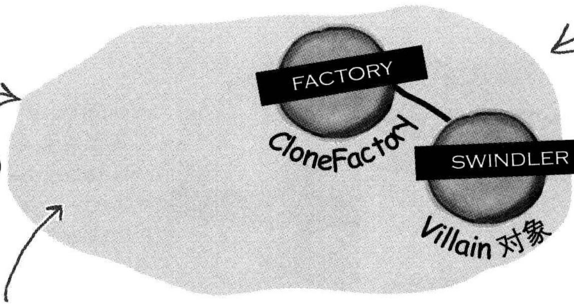
class CloneFactory {
  public Timer SelfDestruct = new Timer();
  public List<object> PeopleInFactory = new List<object>();
  ...
}
```

还有一个Clone类，不过在此也没有显示。回答这些问题不需要这个类。

要为对象增加标签，显示指向它们的引用变量。

我们已经为你完成了第一步。一定要画出显示体系结构的线，这里从克隆工厂到Villain对象画了一条线，因为工厂有它的一个引用（通过其PeopleInFactory字段）。

❶



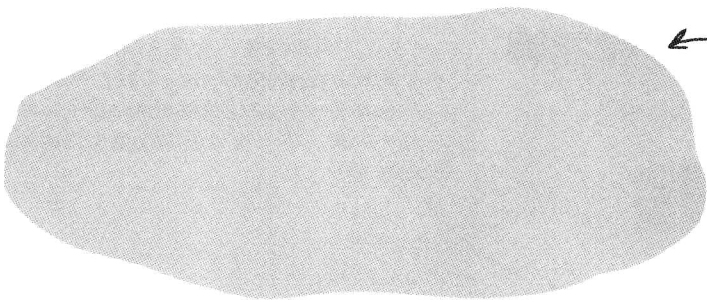

这里留出了空间，因为这一步还要画其他内容。

可以不考虑Clone和List对象，只要增加对应Captain、Swindler、克隆工厂和Swindler逃逸机的对象。

❷



❸



你的任务是画出这两段内存中有什么。

根据你的图，Captain Amazing是在代码中的哪一步殒命的？

..........................................................................................

..........................................................................................

..........................................................................................

..........................................................................................

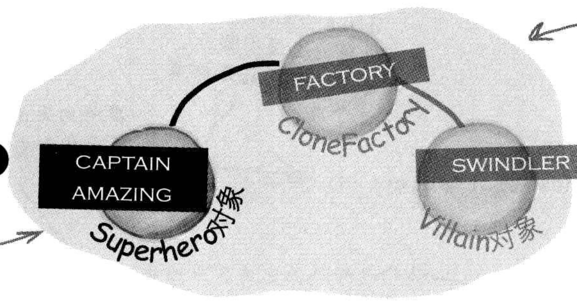
确定后还要在你的图中标出。

# Sharpen your pencil Solution

画出运行FinalBattle程序时内存中的变化。

captainAmazing引用指向一个Superhero对象，swindler引用指向一个Villain对象，克隆工厂的PeopleInFactory列表包含了这两个对象的引用。

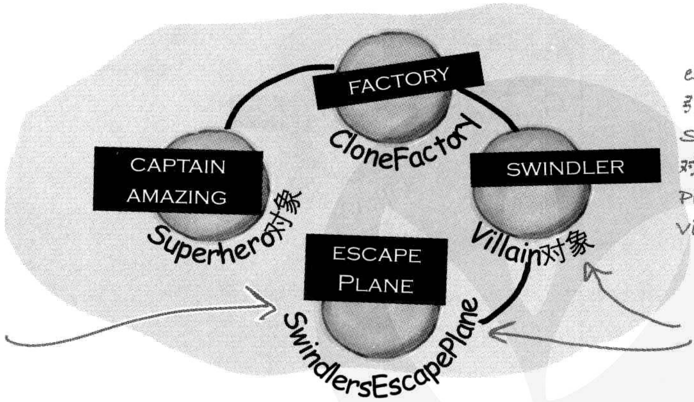
1




要向图中增加这个对象。

escapePlane引用现在指向SwindlersEscapePlane对象的一个新实例，其PilotSeat字段指向一个Villain对象。

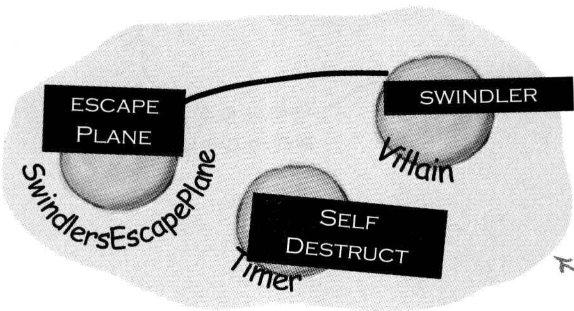
2




只要从escapePlane有指向swindler的引用，swindler就不会被垃圾回收。

selfDestruct触发时，工厂引用变量设置为null，这个引用可以垃圾回收。所以它从这个图中消失。

3




一旦工厂引用消失，会带走CloneFactory对象，这会导致其PeopleInFactory字段引用的List对象引用也消失……而这是保证SuperHero对象活着的惟一引用。下一次垃圾回收器工作时SuperHero将被撤销。

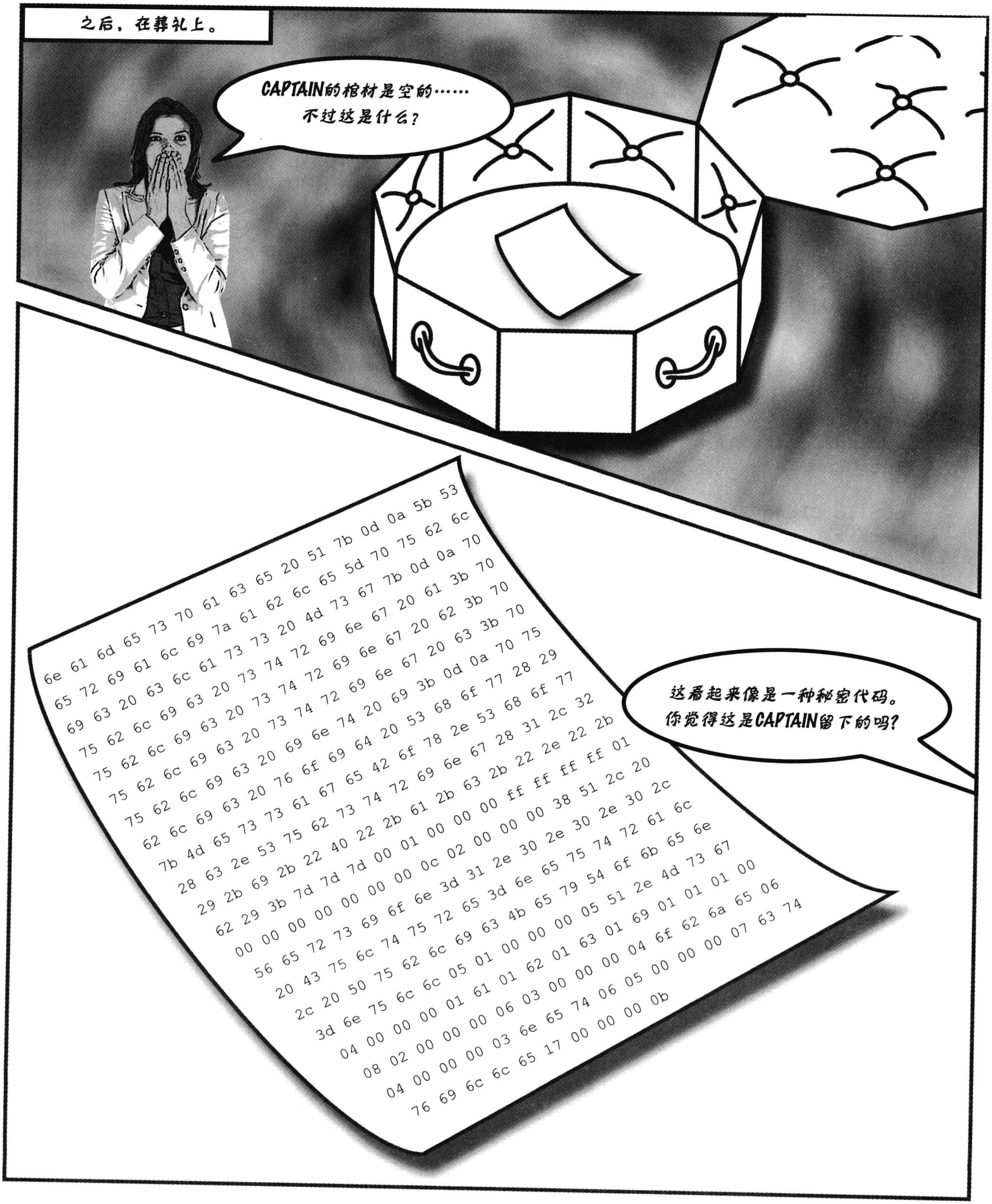
根据你的图，Captain Amazing是在代码中的哪一步殒命的？

```
void SelfDestruct_Tick(object sender, EventArgs e) {
   finalBattle.factory = null;
}
```

FinalBattle构造函数运行一秒之后，英雄离我们而去。

一旦finalBattleFactory设置为null，就可以垃圾回收了……而它会带走对Captain的最后一个引用！

一旦Superhero实例失去了克隆工厂对它的引用，它也会标志为可以垃圾回收。


之后，在葬礼上。
CAPTAIN的棺材是空的……不过这是什么？
6e 61 6d 65 73 70 61 63 65 20 51 7b 0d 0a 5b 53
65 72 69 61 6c 69 7a 61 62 6c 65 5d 70 75 62 6c
69 63 20 63 6c 61 73 73 20 4d 73 67 7b 0d 0a 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 61 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 62 3b 70
75 62 6c 69 63 20 73 74 72 69 6e 67 20 63 3b 70
75 62 6c 69 63 20 69 6e 74 20 69 3b 0d 0a 70 75
62 6c 69 63 20 76 6f 69 64 20 53 68 6f 77 28 29
7b 4d 65 73 73 61 67 65 42 6f 78 2e 53 68 6f 77
28 63 2e 53 75 62 73 74 72 69 6e 67 28 31 2c 32
29 2b 69 2b 22 40 22 2b 61 2b 63 2b 22 2e 22 2b
62 29 3b 7d 7d 7d 00 01 00 00 00 ff ff ff ff 01
00 00 00 00 00 00 00 0c 02 00 00 00 38 51 2c 20
56 65 72 73 69 6f 6e 3d 31 2e 30 2e 30 2e 30 2c
20 43 75 6c 74 75 72 65 3d 6e 65 75 74 72 61 6c
2c 20 50 75 62 6c 69 63 4b 65 79 54 6f 6b 65 6e
3d 6e 75 6c 6c 05 01 00 00 00 05 51 2e 4d 73 67
04 00 00 00 01 61 01 62 01 63 01 69 01 01 01 00
08 02 00 00 00 06 03 00 00 00 04 6f 62 6a 65 06
04 00 00 00 03 6e 65 74 06 05 00 00 00 07 63 74
76 69 6c 6c 65 17 00 00 00 0b
这看起来像是一种秘密代码。你觉得这是CAPTAIN留下的吗？

# 你的最后机会……对象的最终化方法

对象被垃圾回收之前有时需要确保完成一些工作，比如释放一些非托管资源。

对象中有一个特殊的方法，名为最终化方法（finalizer），利用这个方法可以编写撤销对象时执行的代码。可以把它认为是对象个人的finally块：它最后总会执行，而不论前面发生了什么。

下面是Clone类中析构函数的例子：

**一般来讲，如果一个对象只拥有托管资源，你不会为它编写最终化方法。本书中目前为止你看到的都是托管资源—也就是由CLR管理（包括堆上的所有对象）。不过，有时程序员需要访问一个不属于.NET框架的底层Windows资源。如果你在Internet上发现一些使用[DllImport]属性的代码，可能就是在使用一个非托管资源。如果没有以某种方式（可能调用某个方法）"清理"那些非.NET资源，可能会使你的系统不稳定。这正是最终化方法要做的。**

```
[Serializable]
class Clone {
  string Location;
  int CloneID;

  public Clone (int cloneID, string location){
    this.CloneID = cloneID;
    this.Location = location;
  }

  public void TellLocation(string location, int cloneID){
     Console.WriteLine("My Identification number is {0} and " +
                       "you can find me here: {1}.", cloneID, location);
  }

  public void WreakHavoc(){...}

  ~Clone() {
    TellLocation(this.Location, this.CloneID);
    Console.WriteLine ("{0} has been destroyed", CloneID);
  }
}
```

这是构造函数。每次创建一个Clone时都会填入CloneID和Location字段。

这个~(或波浪线)字符表示，这个块中的代码会在对象垃圾回收时运行。

这就是最终化方法。这个不幸的克隆对象向villain发送一个消息，告诉它自己的位置和ID。但是只有当对象垃圾回收时才会运行这部分代码。

编写最终化方法与编写构造函数很类似，不过前面不是一个访问修饰符，而是在类名前放置一个~。这就告诉.NET：垃圾回收对象时应当运行最终化方法块中的代码。

另外，最终化方法不能有参数，因为.NET除了声明"你完了！"并不需要说其他的。




**这里的部分代码只用于学习目的，不要放在实际的程序中。**

本书中我们说过对象"最终"会被垃圾回收，但是从来没有具体指出究竟什么时候回收……只是说有时会在对象引用消失后回收。我们将给出一些代码，在最终化方法中使用GC.Collect()并弹出一个消息框。这些会干扰CLR的工作。这样做只是为了教你有关垃圾回收的知识。如果不是做一些"玩具"性质的程序，千万不要这么做。

# 最终化方法到底何时运行？

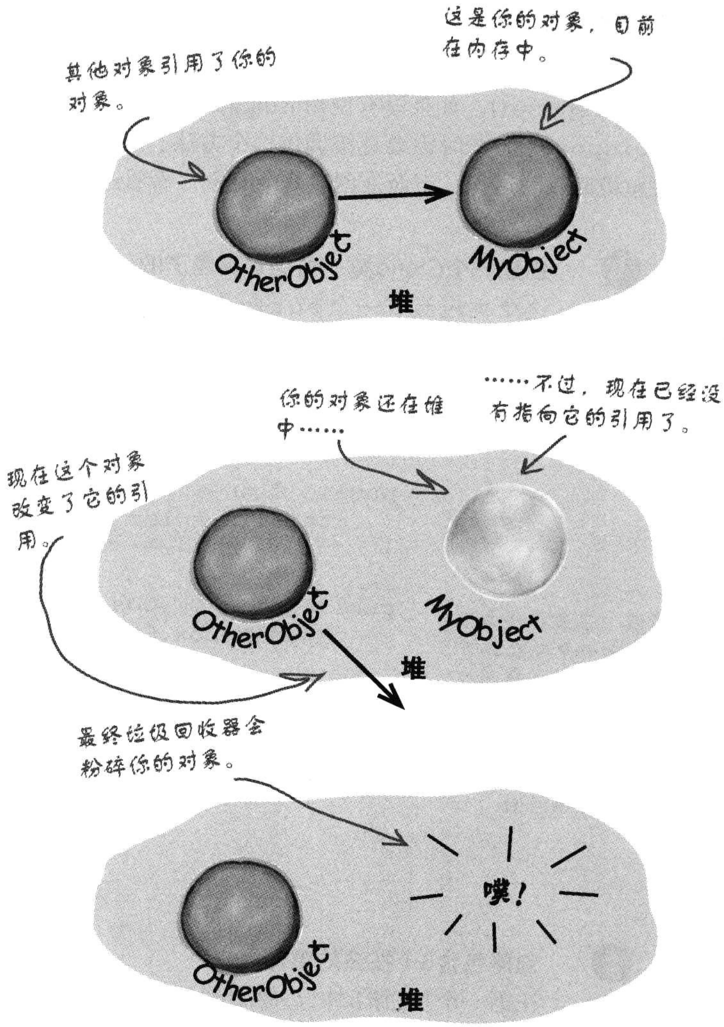
对象的最终化方法在所有引用都消失之后但在对象被垃圾回收之前运行。垃圾回收在对象的所有引用都消失时才发生。不过，并不是引用一消失就进行垃圾回收。

假设你有一个对象，而且有一个引用指向这个对象。.NET要求垃圾回收器工作，它会检查你的对象。但是由于存在指向这个对象的引用，所以垃圾回收器会忽略这个对象，继续检查其他对象。你的对象仍保留在内存中。

然后有情况发生。假设只有最后一个对象保存着一个引用指向你的对象，现在连这个对象也决定改变引用。此时，你的对象仍在内存中，但是没有指向它的引用。现在将无法访问你的对象，这实际上是一个死对象。

不过问题在于，垃圾回收是由.NET控制的，而不是由你的对象来控制。所以，如果几秒甚至几分钟都没有再次运行垃圾回收，你的对象就会一直留在内存中。它是不可用的，但是尚未被垃圾回收。这个对象的最终化方法还没有运行。

最后，.NET再次让垃圾回收器运行。你的最终化方法得以运行…… 而距离删除或改变对象的最后一个引用可能已经过去了几分钟了。这一次，由于你的对象是死对象，所以垃圾回收器会将它消灭。




## 可以建议.NET回收垃圾。

.NET确实允许你建议该回收垃圾了。大多数情况下都不会用到这个方法，因为已经针对CLR中的很多条件对垃圾回收进行了优化，所以直接调用确实不是一个好想法。不过，为了了解最终化方法如何工作，我们可以自己调用垃圾回收。为此，只需调用GC.collect()。

不过，要注意。这个方法并不是要求 .NET立即回收垃圾。它只是说，“尽可能快地回收垃圾”。

```
public void RemoveTheClones(
                 List<Clone> clones) {
  foreach (Clone clone in clonesToRemove)
     clones.Remove(clone);
  GC.Collect();
}
```

如果不是一个“玩具”性质的程序，在其中使用GC.Collect()是一个很糟糕的做法，这一点再强调也不为过，因为这会干扰CLR的垃圾回收器。但这个方法非常适合用来学习有关垃圾回收和最终化方法的知识，所以这里建立一个“玩具”程序来尝试一下。

# Dispose()处理using，最终化方法处理垃圾回收

就像前面看到的一样，即使没有using，Dispose()也能工作。编写Dispose()方法时，不应有任何副作用，不会在运行多次时导致问题。

在一个using语句中创建的对象如果设置为null，或者失去了所有引用，就会运行Dispose()。如果没有使用using语句，只是将引用设置为null并不会调用Dispose()，此时需要直接调用这个方法。对象的最终化方法在对特定对象垃圾回收时运行。下面来创建几个对象，看看这两个方法有什么区别：

动手做！

**1 创建一个Clone类，确保它实现了IDisposable。**

这个类应当有一个int自动属性，名为ID。它有一个构造函数，一个Dispose()方法和一个最终化方法：

```
class Clone : IDisposable {
    public int Id { get; private set; }

    public Clone(int Id) {
        this.Id = Id;
    }

    public void Dispose() {
        MessageBox.Show("I've been disposed!",
                   "Clone #" + Id + " says...");
    }

    ~Clone() {
        MessageBox.Show("Aaargh! You got me!",
                   "Clone #" + Id + " says...");
    }
}
```

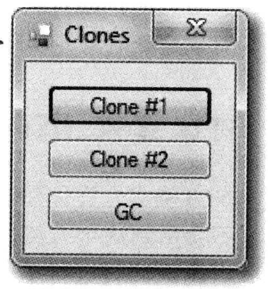
由于这个类实现了IDisposable，它必须有一个Dispose()方法。

这是最终化方法。对象垃圾回收时会运行这个方法。

提醒：在最终化方法中弹出一个消息框可能会扰乱CLR的工作。如果不是一个用来学习垃圾回收知识的"玩具"性质的程序，千万不要这么做。

**2 创建包含3个按钮的窗体。**

你要创建这个窗体。




在第一个按钮的Click事件处理程序中用using语句创建Clone的一个实例。下面是这个按钮相应代码的第一部分：

```
private void clone1_Click(object sender, EventArgs e) {
    using (Clone clone1 = new Clone(1)) {
        // Do nothing!
    }
}
```

这个方法创建一个新的Clone，然后去除它的引用将其立即"杀死"。

由于使用一个using语句来声明clone1，所以会运行它的Dispose()方法。

using块一结束，Clone对象的Dispose()方法就会调用，由于再没有指向它的引用，所以被标志为可以垃圾回收。

**3** **实现另外两个按钮。**

在第二个按钮的Click事件处理程序中创建Clone的另一个实例，并手动将它设置为null：

```
private void clone2_Click(object sender, EventArgs e) {
    Clone clone2 = new Clone(2);
    clone2 = null;
}
```

由于这里没有使用using语句，所以不会运行Dispose()，但是会运行最终化方法。

对于第三个按钮，增加一个GC.Collect()调用来建议进行垃圾回收。

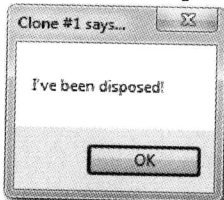
```
private void gc_Click(object sender, EventArgs e) {
    GC.Collect();
}
```

这会建议运行垃圾回收。

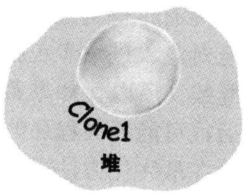
要记住，这样做通常并不是一个好的想法。不过在这里是允许的，因为这样可以很好地了解垃圾回收。

**4** **运行程序，具体调用Dispose()和最终化方法。**

点击第一个按钮并检查消息框：可以看到Dispose()最先运行。




尽管Clone1对象设置为null并运行其Dispose方法，但它还在堆中等待垃圾回收。



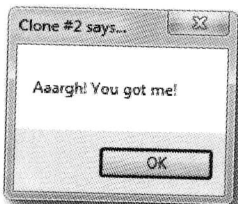

终于……垃圾被回收。大多数情况下，不会看到垃圾回收消息框。因为尽管你的对象设置为null，但垃圾回收尚未运行。

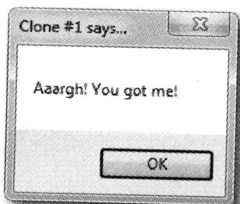
现在点击第二个按钮……什么也没有发生，对不对？这是因为我们没有使用using语句，所以不会调用Dispose()方法。而且在垃圾回收器运行之前，不会看到最终化方法弹出的消息框。

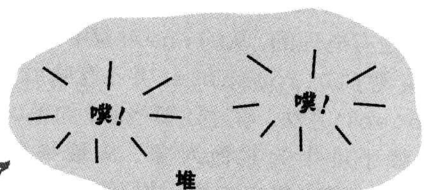
现在Clone2也在堆中，不过没有任何引用指向它。




现在点击第三个按钮，建议完成垃圾回收。你会看到Clone1和Clone2的最终化方法都会触发，并显示相应的消息框。




运行GC.Collect()时，两个对象都会运行其最终化方法并消失。

**尝试运行这个程序。**点击Clone #1按钮，然后点击Clone #2按钮，再点击GC按钮。多做几次。有时Clone #1先回收，有时Clone #2先回收。另外，有时即使没有使用GC.Collect()要求运行垃圾回收，垃圾回收器也会运行。

# 最终化方法<u>不能</u>依赖于稳定性

编写最终化方法时，不能指望它会在某个时间运行。即使调用了`GC.Collect()`（一般应该避免直接调用，除非有充分的理由），这也只是建议运行垃圾回收器，不能保证立即进行垃圾回收。而且垃圾回收时，你也无法知道以怎样的顺序回收对象。

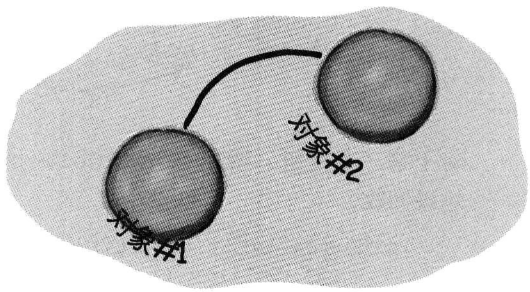
从实际的角度来讲这到底是什么意思呢？可以这样来考虑，如果有两个对象，相互之间有引用。如果对象#1先回收，对象#2指向它的引用就会指向一个不存在的对象。但是如果对象#2先回收，那么对象#1中的引用就会无效。所以这表示，对象的最终化方法不能依赖于引用。也就是说，如果一个最终化方法中所做的工作依赖于存在某些有效的对象引用，这绝对是一个糟糕的做法。

有些事情不适合在最终化方法中完成，比如串行化就是一个很好的例子。如果你的对象有大量其他对象的引用，串行化的前提是所有这些对象都在内存中……还包括它们引用的所有对象，进一步包括这些对象引用的所有对象，如此继续。所以，如果希望在垃圾回收时完成串行化，最后很可能丢失程序的一些重要部分，因为有些对象可能在最终化方法运行前就已经回收了。

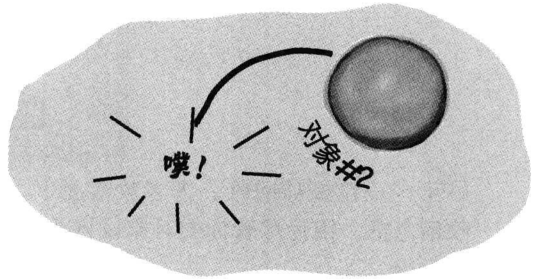
幸运的是，C#对此提供了一个很好的解决方案：`IDisposable`。如果你的工作要修改核心数据，或者依赖于内存中的其他对象，这些工作都要作为`Dispose()`方法的一部分，而不是放在最终化方法中。

有人喜欢把最终化方法认为是`Dispose()`方法的安全防线。这是有道理的，从`Clone`对象中就可以看到，如果只是实现了`IDisposable`，并不意味着会调用对象的`Dispose()`方法。不过还要当心，如果`Dispose()`方法依赖于堆中的其他对象，从最终化方法调用`Dispose()`可能会带来麻烦。最好的解决办法是确保创建`IDispoable`对象时总是使用一个`using`语句。

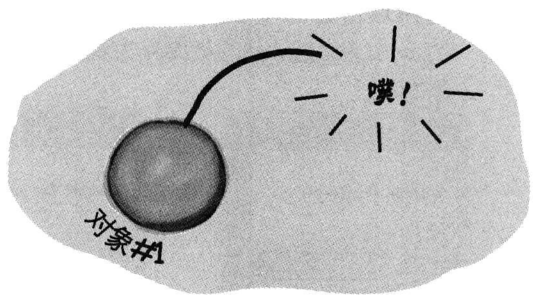
**假设有两个相互引用的对象……**




**……如果它们同时标志为可以垃圾回收，对象#1 可能先消失……**




**……另一方面，对象#2也可能在对象#1之前消失。你无法知道具体的顺序……**




**……正是因为这个原因，一个对象的最终化方法不能依赖于其他对象是否仍在堆中。**

# 让对象在Dispose()中自行串行化

一旦理解了Dispose()和最终化方法之间的差别，要编写在撤销时能自动完成串行化的对象就相当容易了。

动手做！

1. **设置Clone类为可串行化（Serializable）。**

只需在类的前面增加Serializeable属性就可以将它串行化到一个文件。

```
[Serializable]
class Clone : IDisposable
```

2. **修改Clone的Dispose()方法，自行串行化到一个文件。**

下面在Dispose()中使用一个BinaryFormatter将Clone写到一个文件：

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// existing code

public void Dispose() {
  string filename = @"C:\Temp\Clone.dat";
  string dirname = @"C:\Temp\";
  if (File.Exists(filename) == false) {
    Directory.CreateDirectory(dirname);
  }
  BinaryFormatter bf = new BinaryFormatter();
  using (Stream output = File.OpenWrite(filename)) {
    bf.Serialize(output, this);
  }
  MessageBox.Show("Must...serialize...object!",
                  "Clone #" + ID + " says...");
}
```

为访问我们将使用的I/O类，需要更多using语句。

Clone会创建C:\Temp目录，并将自身串行化到一个名为Clone.dat的文件。

这里文件名是硬编码的，我们把文件名作为字符串字面量包含在代码中。对于这样一个小的玩具型程序来说这是可以的，但也不是没有问题。你能想出这可能会导致什么问题，又该如何避免呢？

这个Dispose()方法真的没有副作用吗？如果调用多次会出现什么情况？实现IDisposable时这些问题你都需要考虑。

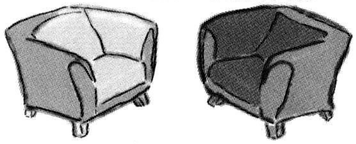
3. **运行应用。**

你看到的表现应该与前几页上完全相同……不过，在Clone1对象垃圾回收前，它会串行化到一个文件。查看这个文件，会看到对象的二进制表示。

**BRAIN POWER**

你认为SuperHero对象的其余代码会是什么样？我们在650页上显示了它的部分代码。你能写出其余的代码吗？

# Fireside Chats



**今晚话题：Dispose()方法和最终化方法正在争论谁更有价值。**

**Dispose()方法：**

老实说，应邀来到这里确实让我有点意外。我认为编程领域早已经有共识了。我的意思是，我肯定比你更有价值。实际上，你确实很没用。甚至不能完成串行化，不能修改核心数据，什么也不能做。而且很不稳定，不是吗？

**最终化方法：**

我没听错吧？真是胡说八道，竟然说我很没用……算了，我不想纠缠这个问题，不过既然谈到这一点……至少我不需要先有一个接口。如果没有IDisposable，你也不过是一个没用的方法。

**Dispose()方法：**

之所以专门有一个接口，那是因为我太重要了。实际上，我是其中唯一的方法！

**最终化方法：**

好吧,好吧……你一直在吹嘘这一点，如果有人实例化对象时忘了使用using语句会怎么样？根本没有你出现的份。

**Dispose()方法：**

你说的没错，程序员必须知道他们需要我，要么直接调用，要么使用一个using语句来调用。不过，他们总能知道我什么时候运行，而且可以用我来完成任何对象清理工作。我很强大，很可靠，而且易于使用。我有这三大功能，你呢？没有人知道你什么时候运行，也不知道你最后运行时应用会是什么状态。

**最终化方法：**

句柄是程序运行时使用的一种资源。.NET和CLR会直接与Windows交互。由于.NET不知道句柄，所以不能为你完成清理。

虽是这么说，但是如果需要在对象垃圾回收的最后那一刻做某件事情，除了我没有别的办法。我会释放网络资源、窗口句柄、流和其他需要清理的资源（如果没有清理这些资源，程序就会出问题）。我可以确保对象更妥善地消失，绝不要小看这一点。

**Dispose()方法：**

这么说来，你能做的我都能做到。就因为你在垃圾回收时运行就自以为是。

**最终化方法：**

喂，我会记住你的这种恶劣态度，等着瞧。




there are no Dumb Questions

**问：最终化方法能使用对象的所有字段和方法吗？**

**答：** 当然. 尽管不能向最终化方法传递参数，但是可以使用对象中的任何字段，可以直接使用，或者通过this来使用，但是要当心，如果这些字段引用了其他对象，那些对象可能已经被回收。不过，确实可以调用最终化对象的其他方法（只要这些方法不依赖于其他对象）。

**问：最终化方法中抛出异常会怎么样？**

**答：** 这个问题问得好。在最终化方法中放一个try/catch块是完全合法的。你可以自己试试看。比如前面写的Clone程序，可以在一个tryp块中创建一个除0异常。捕获这个异常，在前面写的“……I’ve been destroyed”框之前显示一个消息框，指出“I just caught an exception”。现在运行程序，点击第一个按钮，然后点击GC按钮。你会看到异常框和撤销框都会出现（当然，对于不是玩具性质的实际对象，在最终化方法中弹出消息框通常是一个糟糕的想法…… 这些消息框可能根本不应出现）。

**问：垃圾回收器多久自动运行一次？**

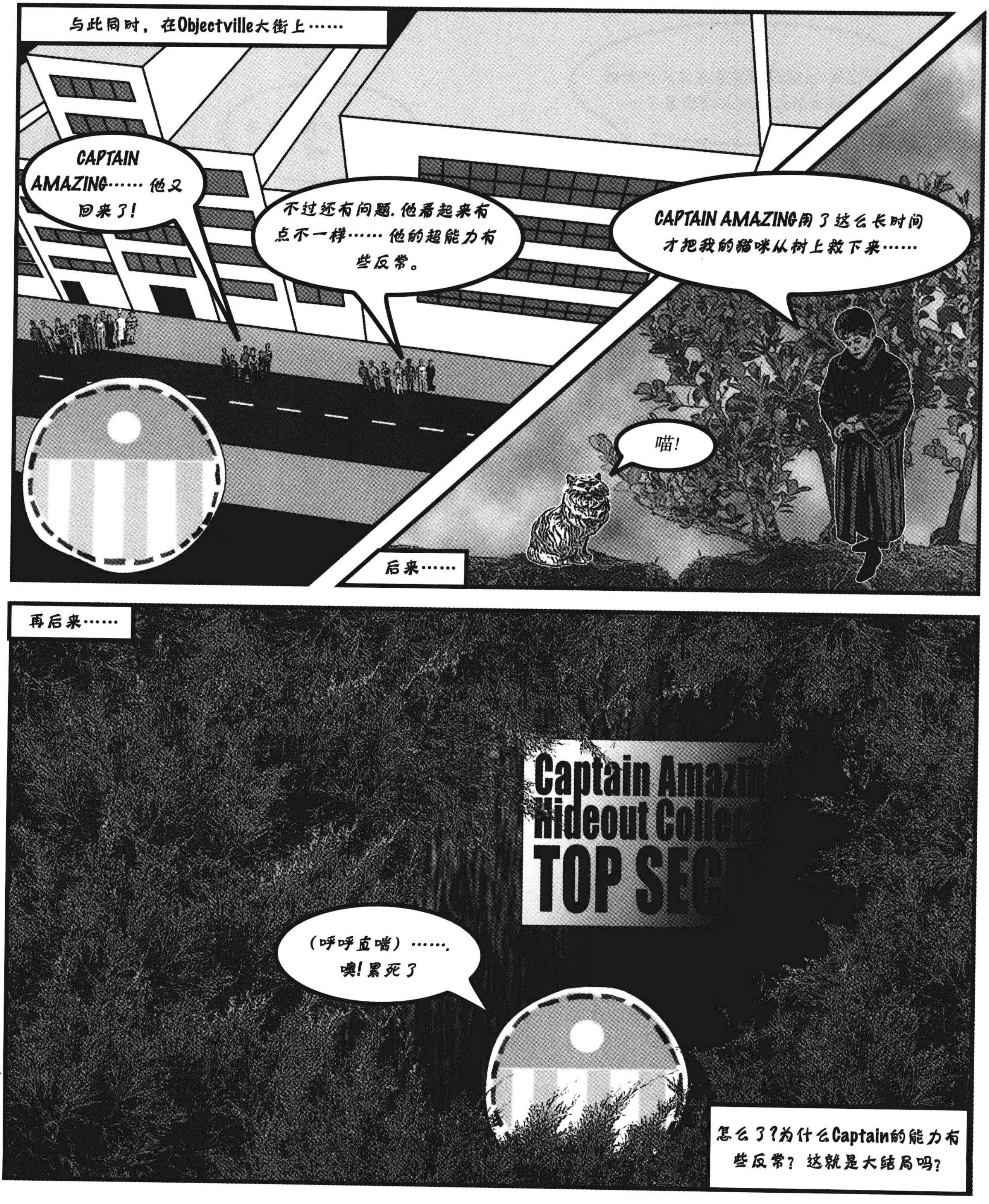
**答：** 这个问题并没有确切的答案。它并不是以一个能轻松预测的周期来运行，而且无法加以控制。可以确信的是，程序退出时它会运行。但是如果希望确保它运行，就必须使用GC.Collect()来提醒…… 即使如此，究竟何时运行还是个未知数。

**问：调用GC.Collect()后，.NET多久以后会启动垃圾回收？**

**答：** 运行GC.Collect()时，就是在告诉.NET尽可能快地回收垃圾。一般是 .NET 一结束它现在的工作就运行垃圾回收。这说明垃圾回收会很快发生，但是至于什么时候发生则无法控制。

**问：如果我确实需要运行某些代码，我把它放在一个最终化方法中，这样对吗？**

**答：** 你的最终化方法有可能不会运行。垃圾回收时有可能禁止运行最终化方法。或者可能完全终止垃圾回收。但是一般来讲都会运行最终化方法。



与此同时，在Objectville大街上……
CAPTAIN AMAZING…… 他又回来了!
不过还有问题，他看起来有点不一样…… 他的超能力有些反常。
CAPTAIN AMAZING用了这么长时间才把我的猫咪从树上救下来……
喵!
后来……
再后来……
Captain Amazin
Hideout Collec
TOP SEC
（呼呼直喘）……，
噢!累死了
怎么了?为什么Captain的能力有些反常? 这就是大结局吗?

# struct看起来像是一个对象……

.NET的另外一些类型我们还没有讨论过，其中就包括struct。struct是结构（structure）的简写，struct看上去与对象非常类似。它们也有字段和属性，这一点与对象相同。而且如果一个方法需要object类型的参数，也可以向这个方法传入一个struct：

```
public struct AlmostSuperhero : IDisposable {
    public int SuperStrength;
    public int SuperSpeed { get; private set; }

    public void RemoveVillain(Villain villain)
    {
        Console.WriteLine("OK, " + villain.Name +
             " surrender and stop all the madness!");
        if (villain.Surrendered)
            villain.GoToJail();
        else
            villain.Kill();
    }

    public void Dispose() { ... }
}
```

Struct可以实现接口，不过不能派生其他类。另外struct是密封的，所以不能作为父类派生子类。

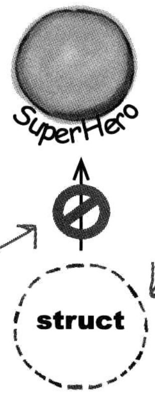
struct可以有字段和属性……

……还可以定义方法。

**对象的强大之处在于它们能通过继承和多态模拟真实世界的行为。**

# ……但不是对象

struct不是对象。它们可以有方法和字段，但是不能有最终化方法。struct不能继承其他类或struct，也不能被其他类或struct继承。




所有struct都继承自System.ValueType，而这进一步继承自System.Object。正是因为这个原因，每个struct都有一个ToString()方法，这是从Object得来的。不过这就是允许struct继承的全部。

struct不能继承其他对象。

可以用struct模仿一个独立对象，但是struct不适合表示复杂的继承体系。

这正是使用类比使用struct多得多的原因。不过这并不意味着struct没有用！

**Struct最适合用来存储数据，但是由于缺乏继承和引用，这可能是一个严重的局限性。**

但是struct与对象最大的差别在于struct是按值复制而不是按引用复制。翻开下一页来了解这是什么意思……

# 复制值，指定引用

类型与类型之间存在着区别，对此你应该有所了解。一方面有一些值类型，如int，bool和string。另一方面还有一些对象类型，如List，Stream和Exception。它们的工作不完全相同，对不对？

下面对值类型和对象类型做一个简要回顾。

用等号把一个值类型变量赋到另一个变量时，会完成值的复制，之后这两个变量相互之间并不连接。另一方面，对引用使用等号时，就会将这两个引用都指向同一个对象。

★ 对于值类型和对象类型，变量声明的做法完全相同。

还记得吧，我们说过方法和语句总放在类中。嗯，看起来这并不是100%正确，它们也可以放在struct中。

int和bool是值类型，List和Exception是对象类型。

```
int howMany = 25;
bool Scary = true;
List<double> temperatures = new List<double>();
Exception ex = new Exception(“Does not compute”);
```

它们都以同样的方式初始化。

★ 但一旦开始赋值时就有所区别了。值类型都是通过复制完成赋值。下面是一个例子：

这一行将存储在fifteenMore变量中的值复制到howMany变量，另外再加15。

修改fifteenMore变量对于howMany没有任何影响，反之亦然。

```
int fifteenMore = howMany;
fifteenMore += 15;
Console.WriteLine(“howMany has {0}, fifteenMore has {1}”,
                             howMany, fifteenMore);
```

这里的输出显示了fifteenMore和howMany相互之间没有连接：

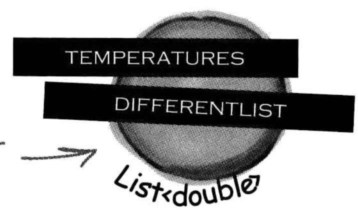
```
howMany has 25, fifteenMore has 40
```

★ 但对于对象赋值，则是在指定引用，而不是具体的值：

这一行将differentList引用设置为指向temperatures引用的同一个对象。

```
temperatures.Add(56.5D);
temperatures.Add(27.4D);
List<float> differentList = temperatures;
differentList.Add(62.9D);
```

两个引用都指向同一个具体对象。




所以修改List意味着两个引用都会看到这个更新…… 因为它们都指向同一个List对象。

```
Console.WriteLine(“temperatures has {0}, differentlist has {1}”,
                          temperatures.Count(), differentList.Count());
```

这里的输出展示了differentList和temperatures实际上指向同一个对象：

```
temperatures has 3, differentList has 3
```

调用differentList.Add()时，它把一个新温度增加到differentList和temperatures所指向的对象。

# Struct是值类型；对象是引用类型

创建struct时，就是在创建一个值类型。这说明，使用等号将一个struct变量设置为等于另一个struct时，会在新变量中创建了这个struct的全新副本。所以尽管struct看上去像是一个对象，但其做法与对象是不同的。

动手做

**1 创建一个名为Dog的struct。**

下面一个简单的struct记录了一只狗的情况。它看上去像是一个对象，但并不是。把它增加到一个新的控制台应用中。

```
public struct Dog {
  public string Name;
  public string Breed;

  public Dog(string name, string breed) {
    this.Name = name;
    this.Breed = breed;
  }

  public void Speak() {
      Console.WriteLine("My name is {0} and I'm a {1}.", Name, Breed);
  }
}
```

没错，这不是一个很好的封装。暂且如此吧，我们只是举个例子。

**2 创建一个名为Canine的类。**

完全复制这个Dog struct，不过将**struct**替换为**class**，然后把**Dog**换成**Canine**（不要忘记重命名Dog的构造函数）。现在就有了一个可以使用的Canine类，这与Dog struct基本上一样。

**3 增加一个按钮，建立Dog和Canine的一些副本。**

下面是Main()方法的代码：

```
Canine spot = new Canine("Spot", "pug");
Canine bob = spot;
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak();

Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak();

Console.ReadKey();
```

你已经在程序中用过struct。还记得第12章和第13章的Point或第9章的DateTime吗？这些都是struct！

Sharpen your pencil

**4 按下按钮前……。**

你认为运行这个代码时的控制台输出是什么，请写下来：

..............................................................................................................

..............................................................................................................

## Sharpen your pencil Solution

你认为写至控制台的输出是什么？

**My name is Spike and I'm a beagle.**

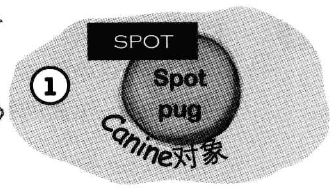
**My name is Jake and I'm a poodle.**

### 发生了什么……

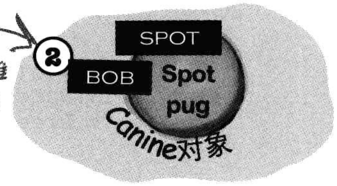
bob和spot引用都指向同一个对象，所以这两个引用会改变相同的字段，并访问同一个Speak()方法。但是struct不是这样。创建betty时，为jake中的数据创建了一个全新的副本。这两个struct彼此是完全独立的。

```
Canine spot = new Canine("Spot", "pug"); ①
Canine bob = spot; ②
bob.Name = "Spike";
bob.Breed = "beagle";
spot.Speak(); ③
```

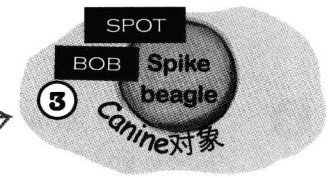
创建了一个新的Canine对象，spot引用指向这个对象。




创建了新的引用变量bob，不过没有向堆增加新对象，bob变量指向spot指向的同一个对象。



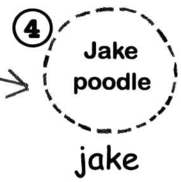

由于spot和bob都指向同一个对象，spot.Speak()和bob.Speak()都会调用同一个方法，而且它们都生成相同的输出（"Spike"和"beagle"）。



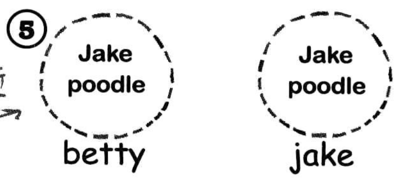

---

```
Dog jake = new Dog("Jake", "poodle"); ④
Dog betty = jake; ⑤
betty.Name = "Betty";
betty.Breed = "pit bull";
jake.Speak(); ⑥
```

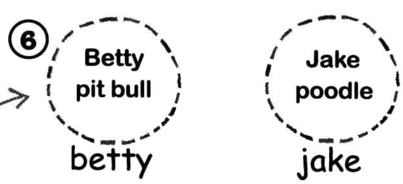
创建一个新struct时，看起来与创建对象很类似，会得到一个变量，可以用这个变量访问它的字段和方法。




这里有一个很大的差别。增加betty变量时，会创建一个全新的值。



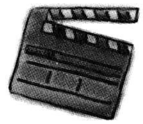

由于创建了数据的一个全新副本，改变betty的字段时，jake不受影响。




**设置一个struct等于另一个struct时，就是在为该struct中的数据创建一个全新的副本。这是因为truct是一个值类型。**

# 栈与堆：再谈内存

**在幕后**



要理解struct与对象有什么区别，这很容易，使用等号会建立struct的一个全新副本，而对象则不同。但是在底层究竟发生了什么？

.NET CLR将内存中的数据划分到两个位置。你已经知道对象放在堆中。CLR还维护着另一部分内存，称为栈，栈用于存储方法中声明的所有局部变量以及传入这些方法的参数。可以把栈认为是能够放入值的一组槽。调用一个方法时，CLR会在栈顶增加更多的槽。返回时则删除这些槽。

要记住，程序运行时，CLR会积极地管理内存，处理堆并完成垃圾回收。

尽管可以将对象变量设置为等于一个struct，但是struct和对象确实是有区别的。

## 代码

**以下是可能在程序中看到的代码。**

## 栈

**struct和局部变量存放在这里。**

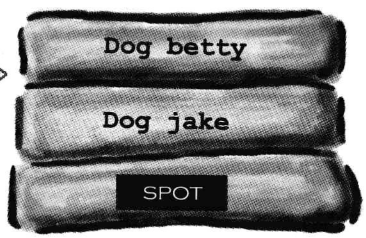
运行这两行代码之后栈的状态。

```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
```




```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
```

创建一个新struct或者任何其他值类型变量时，会向栈增加一个新“槽”。这个槽是该类型值的一个副本。




```
Canine spot = new Canine("Spot", "pug");
Dog jake = new Dog("Jake", "poodle");
Dog betty = jake;
SpeakThreeTimes(jake);

public SpeakThreeTimes(Dog dog) {
    int i;
    for (i = 0; i < 5; i++)
        dog.Speak();
}
```

调用一个方法时，CLR把它的局部变量放在栈顶。方法返回后再从栈顶删除这些局部变量。



等一下，我干嘛需要知道这些？我根本不能直接控制，不是吗？



还可以使用“is”关键字来查看一个对象是否是一个struct或任何其他经过装箱并放在堆上的值类型。

**你肯定想了解按值复制的一个struct与按引用复制的对象有什么不同。**

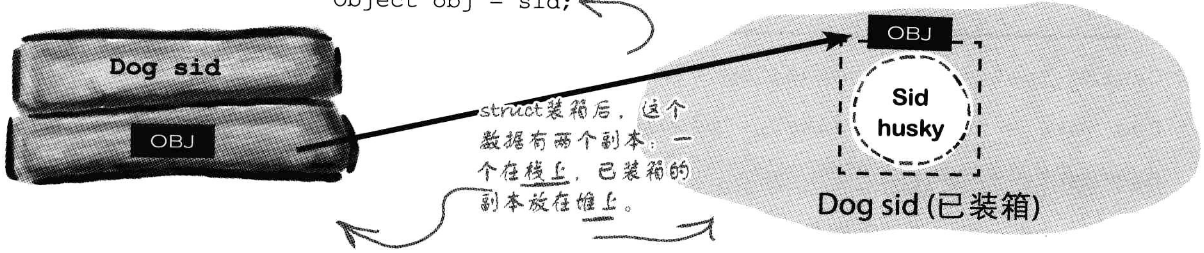
有时可能需要编写一个取值类型或引用类型作为参数的方法，比如说，这个方法要处理Dog struct或Canine对象。如果是这种情况，可以使用object关键字：

```
public void WalkDogOrCanine(object getsWalked) { ... }
```

如果向这个方法发送一个struct，struct就会装箱（boxed）到一个特殊的对象“包装器”中，使它能放在堆中。尽管这个包装器在堆中，但你不能处理struct。必须“解包”（unwrap）为struct才能处理。幸运的是，将一个对象设置为等于一个值类型时，或者将一个值类型传递到本来需要对象的方法时，所有这些装箱和解包都会自动完成。

1. 创建一个对象变量并把它设置为等于一个Dog struct后，栈和堆的状态如下。

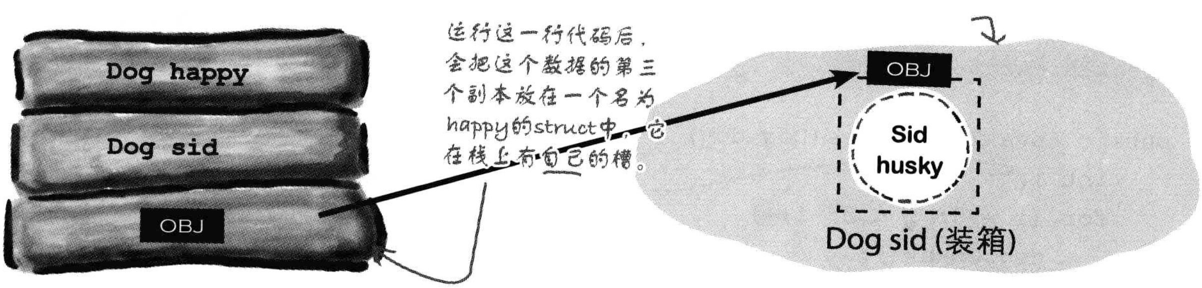
```
Dog sid = new Dog("Sid", "husky");
Object obj = sid;
```




2. 如果想对对象解装（unbox），只需要把它强制转换为正确的类型，就会自动解装。这里不能使用as关键字并提供值类型，所以需要强制转换为Dog。

```
Dog happy = (Dog) obj;
```

这些是struct，所以除非装箱，否则不在堆中。



## 在幕后

**调用一个方法时，它会在栈中寻找参数。**

栈对于CLR如何运行程序扮演着一个重要的角色。有一点我们都承认：可以编写一个方法来调用另一个方法，而后者还可以进一步调用下一个方法。实际上，一个方法还可以调用自身（这称为递归）。正是栈为程序提供了这种能力。

下面是从一个狗模拟程序中选取的几个方法。这些方法都非常简单：FeedDog()要调用Eat()，Eat()则调用了CheckBowl()。

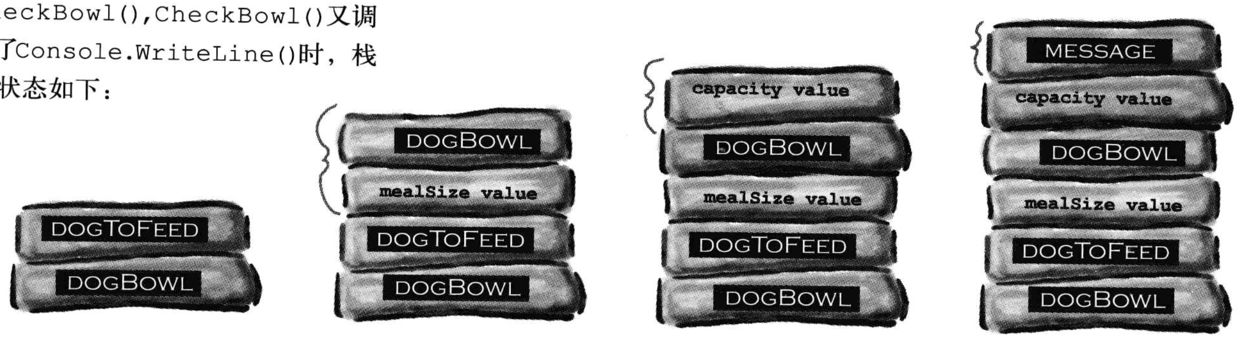
```
public void FeedDog(Canine dogToFeed, Bowl dogBowl) {
    double eaten = Eat(dogToFeed.MealSize, dogBowl);
    return eaten + .05d; // A little is always spilled
}

public void Eat(double mealSize, Bowl dogBowl) {
    dogBowl.Capacity -= mealSize;
    CheckBowl(dogBowl.Capacity);
}

public void CheckBowl(double capacity) {
    if (capacity < 12.5d) {
        string message = "My bowl's almost empty!";
        Console.WriteLine(message);
    }
}
```

记住这里的术语：参数或形参（parameter）是指方法声明中指定所需值的部分；实参（argument）是调用一个方法时传入的具体值或引用。

FeedDog()方法调用了Eat()，而Eat()进一步调用了CheckBowl()，CheckBowl()又调用了Console.WriteLine()时，栈的状态如下：




1. FeedDog()方法有两个参数，一个Canine引用和一个Bowl引用。所以调用这个方法时，传入的两个实参在栈上。
2. FeedDog()需要把两个实参传入Eat()方法，所以也要压入到栈中。
3. 方法调用"堆积"时，程序会越来越深入，因为方法会调用其他方法，而其他方法还会进一步调用另外的方法，此时栈就会越来越大。
4. Console.WriteLine()退出时，它的实参会从栈中弹出。这样一来，Eat()可以继续，就好像什么也没有发生过一样。这正是栈如此有用的原因！

# 用out参数使方法返回多个值

动手做！

说到参数（形参）和实参，还有一些方法可以为程序传入值和由程序返回值，这些方法都需要向方法声明增加修饰符。其中最常用的方法之一是使用out修饰符指定一个输出参数。做法如下。创建一个新的Windows Forms应用，为窗体增加以下空方法声明。注意两个参数的out修饰符：

```
public int ReturnThreeValues(out double half, out int twice)
{
        return 1;
}
```

通过使用out参数，方法可以返回多个值。

编译代码时，你会看到两个错误：out参数‘half’必须在控制离开当前方法之前得到赋值（对于‘twice’参数也会看到一个同样的错误消息）。只要使用out参数，必须在方法返回之前设置这个参数，这就像如果方法声明有一个返回值，总是要使用一个return语句一样。整个方法如下：

```
Random random = new Random();
public int ReturnThreeValues(out double half, out int twice) {
    int value = random.Next(1000);
    half = ((double)value) / 2;
    twice = value * 2;
    return value;
}
```

这个方法在返回之前必须设置所有out参数，否则将无法编译。

既然已经设置了两个out参数，它应该能编译。现在来使用这些返回值。增加一个按钮，按钮的事件处理程序如下：

```
private void button1_Click(object sender, EventArgs e) {
    int a;
    double b;
    int c;
    a = ReturnThreeValues(b, c);
    Console.WriteLine("value = {0}, half = {1}, double = {2}", a, b, c);
}
```

注意到了吗？并不需要初始化b和c。如果将一个变量用作为一个out参数的实参，则无需初始化这个变量。

唉呀！出现了更多编译错误：传入参数1时必须带out关键字。每次调用一个有out参数的方法时，传入实参时都要使用out关键字。这行代码应该如下所示：

```
a = ReturnThreeValues(out b, out c);
```

现在你的程序将顺利编译。运行时，ReturnThreeValues()方法会设置3个值，并返回所有3个：a是方法的返回值，b是half参数返回的值，c是twice参数返回的值。

# 使用ref修饰符按引用传递

有一点你已经反复见过，每次向方法传入一个int、double、struct或任何其他值类型时，都是在向这个方法传入这个值的一个副本。对此有一个名字，称之为按值传递或传值，这表示复制整个参数值。

不过向方法传递实参还有一种方法，这称为按引用传递或传引用。可以使用ref关键字允许方法直接处理传入的实参。就像out修饰符一样，声明方法时要使用ref，调用时也要使用ref。这是一个值类型还是一个引用类型并不重要—传入方法ref参数的任何变量都可以由该方法直接修改。

实际上，out参数就像一个ref参数，只不过进入方法之前out参数无需赋值，但是在方法返回之前必须赋值。

可以看看这是如何做的—把这个方法增加到你的程序中：

```
public void ModifyAnIntAndButton(ref int value, ref Button button) {
    int i = value;
    i *= 5;
    value = i - 3;
    button = button1;
}
```

这个方法设置value和button参数时，实际上会修改调用它的button2_Click()方法中q和b变量的值。

再增加一个按钮，它的事件处理程序要调用这个方法：

```
private void button2_Click(object sender, EventArgs e) {
    int q = 100;
    Button b = button3;
    ModifyAnIntAndButton(ref q, ref b);
    Console.WriteLine("q = {0}, b.Text = {1}", q, b.Text);
}
```

这会打印"q = 497, b.Text = button1"因为这个方法实际上修改了q和b变量。

button2_Click()调用ModifyAnIntAndButton()方法时，按引用传入其q和b变量。ModifyAnIntAndButton()方法会像使用任何其他变量一样使用这两个变量。不过，由于它们是按引用传递的，这个方法实际上一直在更新q和b变量，而不是更新它们的一个副本。所以这个方法退出时，q和b变量会更新为修改后的值。

运行程序进行调试，为q和b变量增加监视来看这是如何做的。

## 内置值类型的TryParse()方法使用out参数

对于out参数有一个极好的例子，一些内置的值类型就使用了out参数。我们经常希望将类似"35.67"的字符串转换为一个double。有一个方法可以做到这一点：double.Parse("35.67")会返回double值35.67。但是double.Parse("xyz")会抛出一个FormatException异常。有时这正是你想要的，但是还有一些情况下你可能希望检查一个字符串是否可以解析为一个值。这里就可以用到TryParse()方法：double.TryParse("xyz", out d)会返回false，并设置i为0，而double.TryParse("35.67", out d)会返回true，并设置d为35.67。

另外，还记得吗？我们在第9章使用了一个switch语句将"Spades"转换为Suits.Spades。嗯，除了枚举，静态方法Enum.Parse()和Enum.TryParse()也可以做同样的事情！

# 使用可选参数设置默认值

很多情况下，调用方法时要反复提供同样的实参，不过这个方法还是需要有这个参数，因为有时确实可能有所不同。如果可以设置一个默认值会很有用，这样调用方法时只有当实参不同时才需要指定。

这正是可选参数的作用。在方法声明中可以指定一个可选参数，为此要使用一个等号后面是该参数的默认值。可以根据需要有多个可选参数，但是所有可选参数都必须放在必要参数后面。

下面是一个例子，这个方法使用可选参数检查一个人是否发烧：

```
void CheckTemperature(double temperature, double tooHigh = 99.5, double tooLow = 96.5)
{
    if (temperature < tooHigh && temperature > tooLow)
        Console.WriteLine("Feeling good!");
    else
        Console.WriteLine("Uh-oh -- better see a doctor!");
}
```

这个方法有两个可选参数：tooHigh的默认值为99.5，tooLow默认值为96.5。调用CheckTemperature()时，如果只提供一个实参，就会使用tooHigh和tooLow的默认值。如果调用方法时提供了两个实参，它会使用第二个实参作为tooHigh的值，另外仍使用tooLow的默认值。也可以指定所有3个实参为这3个形参传入值。

还有一种选择。如果希望使用部分（但不是全部）默认值，可以使用命名参数，从而只向你希望传入值的那些参数传入参数值。为此只需要给出各个参数名，后面是一个冒号以及参数值。如果使用多个命名参数，一定要用逗号分隔开，就像其他参数一样。

为窗体增加CheckTemperature()方法，然后增加一个按钮，其事件处理程序如下。进行调试，确保你真正了解它是如何工作的：

```
private void button3_Click(object sender, EventArgs e)
{
    // Those values are fine for your average person
    CheckTemperature(101.3);

    // A dog's temperature should be between 100.5 and 102.5 Fahrenheit
    CheckTemperature(101.3, 102.5, 100.5);

    // Bob's temperature is always a little low, so set tooLow to 95.5
    CheckTemperature(96.2, tooLow: 95.5);
}
```

如果希望方法有默认值，可以使用可选参数和命名参数。

# 需要不存在的值时可以使用可为空的类型

花点时间翻回第1章回顾一下转换为数据库的联系卡。还记得如何建立数据库表允许各个列为空吗？这样一来，如果有人漏了某个值，或者有些值写得难以辨认，数据库可以使用null来表示这一列没有值。正常情况下，使用null就可以。不过对于struct（和int、boolean以及其他值类型），不能把它们设置为null。如果有下面的语句：

甚至Client也设置为允许为null值，这看上去是不是很奇怪？一个人要么是客户，要么不是，对不对？不过不能保证每张联系卡上的Client空都已填，数据库需要有一种办法来表示我们可能不知道一个人是不是客户的情况。

```
bool myBool = null;
DateTime myDate = null;
```

尝试编译程序时，这些语句就会导致错误！

假设你的程序需要处理一个日期和时间值。正常情况下会使用一个DateTime变量。但是如果这个变量不一定有值怎么办？这里就可以用到可为空（nullable）的类型。只需要在值类型最后增加一个问号（？），它就会变成一个可为空的类型，可以将它设置为null。

```
bool? myNulableInt = null;
DateTime? myNullableDate = null;
```

| Nullable<DateTime> |
|---|
| Value: DateTime<br>HasValue: bool<br>... |
| GetValueOrDefault(): DateTime<br>... |

每个可为空的类型都有一个名为Value的属性，可以获取或设置这个值。DateTime?会有一个类型为DateTime的Value，int?的Value类型为int，依此类推。它们还有一个名为HasValue的属性，如果值不为null则返回true。

Nullable<T>是一个struct，允许你存储一个值类型或者一个null值。这是Nullable<DateTime>的一些方法和属性。

总是可以将一个值类型转换为一个可为空的类型：

```
DateTime myDate = DateTime.Now;
DateTime? myNullableDate = myDate;
```

不过为把可为空的类型赋回一个值类型，需要进行强制转换：

```
myDate = (DateTime) myNullableDate;
```

如果HasValue为false，Value属性会抛出一个InvalidOperationException异常，强制类型转换也同样会抛出这个异常（因为强制转换就等价于使用Value属性）。

## 问号T?是Nullable<T>的别名

为某个值类型增加一个问号时（如int?或decimal?），编译器会把它翻译为Nullable<T> struct（Nullable<int> 或Nullable<decimal>）。你可以自己检查看看：向程序增加一个Nullable<DateTime>变量，加一个断点，再在调试工具中增加一个监视项。你会看到IDE的监视窗口中显示的System.DateTime?。这是别名的一个例子，这并不是你第一次遇到别名。把鼠标停在任意一个int上。你会看到它会翻译为一个名为System.Int32的struct:

int.Parse()和int.TryParse()都是这个struct的成员。

```
int value;
 struct System.Int32
 Represents a 32-bit signed integer.
```

花点时间对第4章最前面列出的各个类型做这个检查。注意它们都是某个struct的别名，除了string以外，这是一个类，名为System.String（这是一个引用类型，而不是值类型）。

# 可为空的类型可以让程序更健壮

用户可能会做各种各样疯狂的事情。你以为你知道人们会怎样使用你写的程序，但事实上有人可能会以你意想不到的顺序点击按钮，或者在一个文本框里输入256个空格，也可能使用Windows任务管理器在数据写入文件进行到一半时退出程序，突然之间会冒出各式各样的错误。第10章我们说过一个程序如果能妥善地处理格式不当、意料之外或者稀奇古怪的输入，就称为是健壮的。是这样，处理用户的原始输入时，可空空的类型对于让程序更为健壮非常有用。现在你来自己试试看，创建一个新的控制台应用，并增加下面这个RobustGuy类：

增加RobustGuy.ToString()时，输入Birthday.Value时请注意智能提示窗口。由于Value属性是一个DateTime，你会看到DateTime的所有成员。

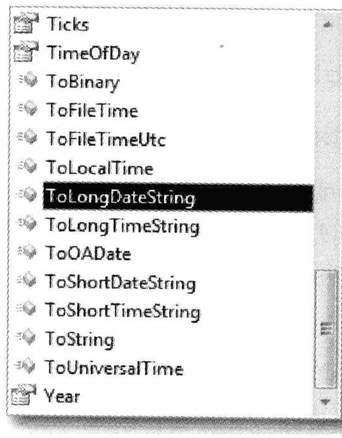
使用ToLongDateString()方法把它转换为一个人可读的字符串。

```
class RobustGuy {
    public DateTime? Birthday { get; private set; }
    public int? Height { get; private set; }

    public RobustGuy(string birthday, string height) {
        DateTime tempDate;
        if (DateTime.TryParse(birthday, out tempDate))
            Birthday = tempDate;
        else
            Birthday = null;

        int tempInt;
        if (int.TryParse(height, out tempInt))
            Height = tempInt;
        else
            Height = null;
    }

    public override string ToString() {
        string description;
        if (Birthday != null)
            description = "I was born on " + Birthday.Value.ToLongDateString();
        else
            description = "I don't know my birthday";
        if (Height != null)
            description += ", and I'm " + Height + " inches tall";
        else
            description += ", and I don't know my height";
        return description;
    }
}
```




使用DateTime和int TryParse()方法尝试将用户输入转换为值。

如果用户输入的是乱七八糟的"垃圾"，Nullable类型就没有值，HasValue()方法将返回false。

尝试其他以"To"开头的DateTime方法，看看对程序的输出有什么影响。

这时程序的Main()方法。它使用**Console.ReadLine()**得到用户的输入：

```
static void Main(string[] args) {
    Console.Write("Enter birthday: ");
    string birthday = Console.ReadLine();
    Console.Write("Enter height in inches: ");
    string height = Console.ReadLine();
    RobustGuy guy = new RobustGuy(birthday, height);
    Console.WriteLine(guy.ToString());
    Console.ReadKey();
}
```

Console.ReadLine()允许用户在控制台窗口中输入文本。用户按回车时，它会将输入作为一个字符串返回。

**运行这个程序时，看看对日期输入不同的值时会发生什么。DateTime.TryParse()可以得出大多数值。输入一个它无法解析的日期时，RobustGuy的Birthday属性就没有值。**

# 池塘谜题



你的任务是从池塘里取出代码片段，在代码中填空。一个代码片段可以使用多次，另外不是所有代码片段都会用到。你的目标是完成这个代码，使得创建Faucet类的一个新实例时向控制台写出以下输出：

```
public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

**创建一个新Faucet对象时的输出：**

**back in 20 minutes**

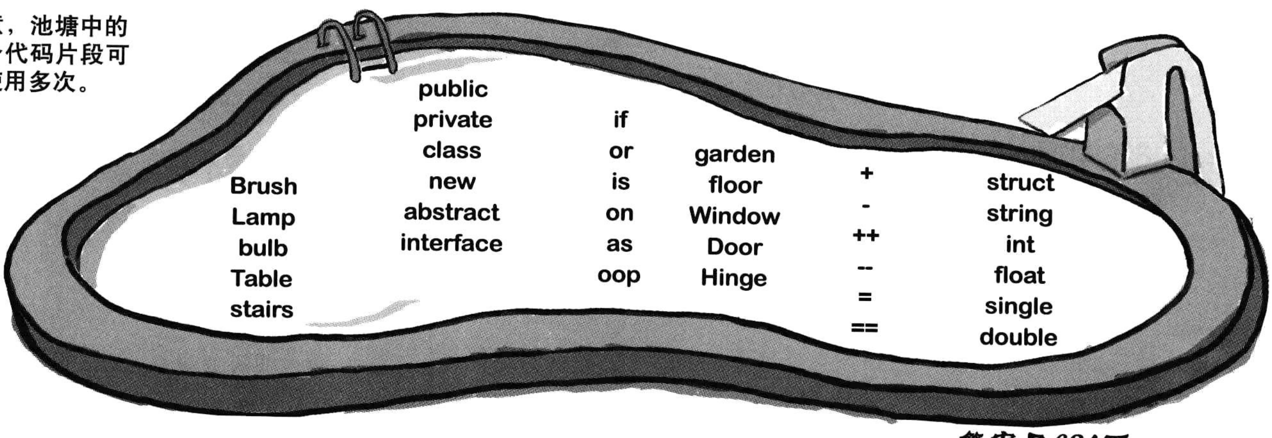
这是我们的目标……得到这个输出

```
public ________ Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil _____ int)
            ________.bulb = (int)oil;
        else if (oil _____ string)
            stairs = (string)oil;
        else if (oil _____ Hinge) {
            ________ vine = oil _____ ________;
            Console.WriteLine(vine.Table()
             + " " + _______.bulb + " " + stairs);
        }
    }
}

public ________ Hinge {
    public int bulb;
    public Table garden;
    public void Set(Table a) {
        garden = a;
    }
    public string Table() {
        return ________.stairs;
    }
}
```

**附加题：圈出发生装箱的代码行。**

注意，池塘中的各个代码片段可以使用多次。




答案见684页。

## there are no Dumb Questions

**问：** **先等一下。为什么要考虑栈？**

**答：** 因为理解栈与堆之间的差别有助于你搞清楚引用类型和值类型。struct和对象的工作是完全不同的，但这一点很容易忘记，对它们使用等号时，看起来很相似。如果对.NET和CLR在底层如何处理有所了解，这能帮助你理解引用类型和值类型之间为什么不同。

**问：** **那么装箱呢？为什么装箱对我来说很重要？**

**答：** 因为你需要知道数据什么时候放在栈里，另外需要知道数据什么时候会来回复制。装箱需要占用内存，而且需要更多时间。如果程序中只完成几次装箱（或者几百次），可能注意不到差别。但是假设你写的程序要反复做这件事呢？可能每秒上百万次。这种情况并不是遥不可及，因为你的蜂窝模拟系统就是如此。如果发现程序占用越来越多的内存，或者越来越慢，可以考虑在程序中重复的部分避免装箱来提高效率。

**问：** **我明白了，把一个struct变量设置为等于另一个struct时会得到这个struct的一个全新副本。但是这对我有什么用？**

**答：** 这对于封装很有用。来看下面这个我们很熟悉的代码（取自一个了解自己位置的类）：

```
private Point location;
public Point Location {
  get { return location; }
}
```

如果Point是一个类，这就是一个很糟糕的封装。即使location是私有的也无济于事，因为这里建立了一个公共的只读属性，可以返回它的一个引用，所以任何其他对象都能访问它。

好在，Point实际上是一个struct。这说明公共Location属性只返回这个点的一个全新副本。使用这个属性的对象可以对这个副本做任何处理，但这些改变不会影响私有location字段。

**问：** **既然Point是一个struct，这是不是说明我还用过其他的struct？**

**答：** 没错！有一个非常有用的struct，在处理图形化和窗体时很常用，这就是Rectangle。它有一些非常有用的方法，如果需要确定边界，以及检查一个点是否在矩形以内（或以外），就可以使用这些方法。只需设置其位置和大小，它会自动计算top、bottom、left、right、width和height值。

你还用过另一个有用的struct：Size。前面已经见过它的具体使用，使用MeasureString()方法确定一个串的大小时就用到过这个struct。要知道这也是一个struct。

**问：** **怎么确定使用一个struct还是使用一个类呢？**

**答：** 大多数情况下，程序员都会使用类。struct有很多限制，所以用来完成大型任务时可能很困难。它们不支持继承、抽象或多态，而且你已经知道了这些对于轻松构建程序是多么重要。

但有些方面struct会很有用，如果数据的类型有限，而且数据量很少，但需要反复处理，就可以使用struct。Rectangle和Point就是很好的例子——对它们不会做太多处理，不过会反复地使用。struct往往比较小，而且很受限。如果有少量不同类型的数据，希望保存在一个类中的一个字段中，或者希望作为参数传入一个方法，这就非常适合使用struct。

**希望为类提供很好的封装性时，struct很有意义，因为返回一个struct的只读属性会建立它的一个全新副本。**

### Sharpen your pencil

这个方法想杀死一个Clone对象，但是没有办到。为什么没有成功？

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

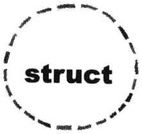
..............................................................................

..............................................................................

热点问题！答案见678页。

# Captain Amazing…… 还不够

有了以上对装箱的介绍，你应该已经很清楚这个能力欠佳，容易疲劳的Captain　Amazing到底怎么了。实际上，它根本不是原来的Captain Amazing，这只是一个装箱的struct:




**vs.**




这是struct（和其他值类型）的一大优点，可以很容易地建立值类型的副本。

**struct**

1. **struct不能继承类或实现接口。**
   难怪Captain的超能力显得那么弱！他没有继承任何行为。

2. **struct按值复制。**
   这是struct最有用的一点。对于封装尤其有意义。

**SuperHero**

1. **不能创建对象的全新副本。**
   设置一个对象变量等于另一个变量时，只是复制了对同一个变量的引用。

2. **对对象可以使用“as”关键字。**
   对象允许多态，允许一个对象表现为它继承的某个对象。



# 扩展方法为现有类增加新行为

还记得第7章的sealed访问修饰符吗？利用它可以建立一个不能扩展的类。

有时需要扩展一个不能继承的类，如一个密封类（sealed class），很多.NET类都是密封类，所以不能由它们派生子类。C#为此提供了一个强大的工具：扩展方法（extension method）。向工程增加一个包含扩展方法的类时，会为已有的类增加新方法。所要做的就是创建一个静态类，并增加一个静态方法，使用this关键字将要扩展的类的一个实例作为它的第一个参数。

假设有一个`sealed OrdinaryHuman`类（记住，这说明不能扩展这个类）：

```
sealed class OrdinaryHuman {
    private int age;
    int weight;

    public OrdinaryHuman(int weight){
        this.weight = weight;
    }

    public void GoToWork() { /* code to go to work */ }
    public void PayBills() { /* code to pay bills */ }
}
```

OrdinaryHuman类是密封类，所以不能派生子类。但是如果想为它增加一个方法该怎么做呢？

增加扩展方法时，使用"this"关键字指定第一个参数。

由于我们希望扩展OrdinaryHuman类，所以让第一个参数是this OrdinaryHuman。

`SuperSolierSerum`方法向`OrdinaryHuman`增加一个扩展方法：

```
static class SuperSoldierSerum {
    public static string BreakWalls(this OrdinaryHuman h, double wallDensity) {
        return ("I broke through a wall of " + wallDensity + " density.");
    }
}
```

扩展方法总是静态方法，而且必须放在静态类中。

一旦将`SuperSoldierSerum`类增加到工程中，`OrdinaryHuman`就得到一个`BreakWalls`方法。所以现在窗体可以使用这个方法：

```
static void Main(string[] args){
    OrdinaryHuman steve = new OrdinaryHuman(185);
    Console.WriteLine(steve.BreakWalls(89.2));
}
```

程序创建OrdinaryHuman类的一个实例时，可以直接访问BreakWalls()方法（只要它能访问这个SuperSoldierSerum类）。

继续自己试试看！创建一个新的控制台应用，增加这两个类和Main()方法。调试进入BreakWalls()方法，看看做了什么。

## Sharpen your pencil Solution

这个方法本来要杀死一个Clone对象，但是没有办到，为什么没有成功？

```
private void SetCloneToNull(Clone clone) {
    clone = null;
}
```

这个方法所做的只是设置其参数为null，但是这个参数只是一个Clone的引用。就像在一个对象上粘一个标签，然后再把它撕掉一样。

所以clone参数只在栈上，因此把它设置为null不会对堆有任何影响。

## there are no Dumb Questions

**问：请你再解释一下，为什么要使用扩展方法，为什么不能把我需要的新方法直接增加到类代码中？**

答：这么做是可以的，而且如果只是讨论向一个类增加方法，可能就应该这样做。扩展方法用得很少，而且只在特殊情况下才使用，即在出于某种原因不能修改所扩展的类时（如这是.NET Framework或另一个第三方库的一部分），才会使用扩展方法。如果需要扩展一个类的行为，而正常情况下本来不能访问这个类（如.NET框架或另一个库免费提供的一个类型或对象），扩展方法会非常有用。

**问：为什么要使用扩展方法？为什么不利用继承扩展这个类呢？**

答：如果可以扩展这个类，通常就会这样做，扩展方法并不是要完全取代继承。不过，如果类不允许扩展，扩展方法就会很方便。利用扩展方法，可以改变整个一组对象的行为，甚至可以向.NET Framework中最基本的类增加功能。扩展类可以提供新行为，但是如果你想使用这个新行为，就要使用新子类。

**问：扩展方法会影响一个类的所有实例，还是只影响这个类的某个特定实例？**

答：它会影响所扩展的类的所有实例。实际上，一旦创建一个扩展方法，这个新方法就会与所扩展类的正常方法一同出现在IDE中。

对于扩展方法还有一点要记住：不能通过建立扩展方法来访问类的任何内部细节，所以扩展方法仍相当于“外来户”！






**完全正确！有些类是不能继承的。**

打开任意一个工程，增加一个类，键入以下代码：

```
class x : string { }
```

现在试着编译你的代码，IDE会报错。这是因为有些.NET类是密封的，这说明不能从继承这些类（你也可以试试你自己的类！只要在public访问修饰符后面为类增加sealed关键字，其他类就不能从它继承了）。尽管不能继承这样一个类，但扩展方法提供了扩展这种类的一个途径。

不过，扩展方法的作用还不仅如此。除了扩展类，还可以扩展接口。只需使用一个接口名取代类名，放在扩展方法第一个参数的this关键字后面。这样一来，这个扩展方法就会增加到实现了这个接口的每一个类。还记得第12章增加到模拟系统的LINQ代码吗？LINQ就是完全利用扩展方法构建的，它扩展了 IEnumerable<T> 类（第15章中将更多地了解LINQ）。

这方面也无法使用继承，没有办法继承一个接口。

# 扩展一个基本类型：string

通常我们不会改变一个语言中最基本的类型的行为，如string。不过，利用扩展方法，确实可以扩展这些类型！创建一个新工程，增加一个名为HumanExtensions.cs的文件。

动手做！

**1** **把所有扩展方法放在一个单独的命名空间。**

把所有扩展方法放在一个单独的命名空间，与其他代码分开，这是一个很好的想法。这样一来，就不必操心查看其他程序中是否使用了同名方法。还要建立一个静态类，来放置所增加的方法。

```
namespace MyExtensions {
  public static class HumanExtensions {
```

使用一个单独的命名空间有利于代码的组织。

定义扩展方法的类必须是一个静态类。

**2** **创建静态扩展方法，并定义第一个参数为this，然后是所扩展的类型。**

声明一个扩展方法时需要知道重要的两点：方法必须是静态的，另外它扩展的类要作为第一个参数。

```
public static bool IsDistressCall (this string s){
```

"this string"表示我们要扩展string类。

扩展方法也必须是静态的。

**3** **把处理串的代码放在这个方法中。**

```
public static class HumanExtensions {
  public static bool IsDistressCall(this string s){
    if (s.Contains("Help!"))
      return true;
    else
      return false;
  }
}
```

你希望其他命名空间中的代码能访问这个类，所以一定要把它标志为public！

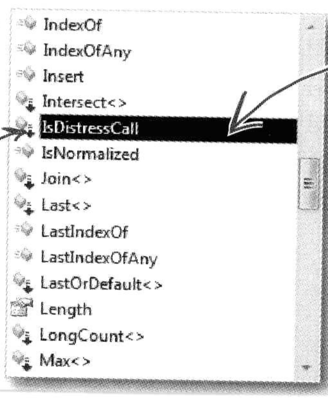
这会检查串中是否有某个特定的值……这个功能绝对是默认string类中没有的。

**4** **创建一个窗体，并增加一个串。**

在窗体代码最前面增加using MyExtensions;，为窗体增加一个按钮，从而可以在这个按钮的事件处理程序中尝试这个新的扩展方法。现在，使用一个串时就会得到这些扩展方法可以自由使用。可以键入一个string变量名和一个点号，自己试试看：

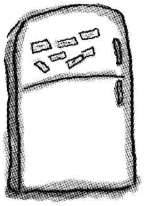
```
string message = "Clones are wreaking havoc at the factory. Help!";
message.
```

一旦键入点号，IDE就会弹出一个帮助窗口，列出string的所有方法……其中也包括你的扩展方法。




把using行注释掉，这个扩展方法会从智能提示窗口消失。

**这个小例子只是展示了扩展方法的语法。下一章将会真正了解扩展方法的巨大作用。其中主要讨论LINQ，它就是完全利用扩展方法实现的。**



# 扩展磁条

组织这些磁条，生成以下输出：

**a buck begets more bucks**

```
namespace Upside {
    public static class Margin {
        public static void SendIt

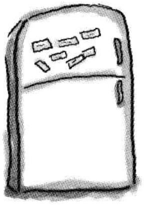
        }
        public static string ToPrice

    }
}
}
}
```

```
using Upside;
namespace Sideways {
    class Program {

        Console.ReadKey();
        }
    }
}
```

```
s.SendIt();
i = 3;
bool b = true;
if (b == true)
    return "be";
Console.Write(s);
public static string Green
if (n == 1)
    return "a buck ";
b.Green().SendIt();
.SendIt();
(this string s) {
b.Green().SendIt();
b = false;
else
    return " more bucks";
else
    return "gets";
i.ToPrice()
(this bool b) {
int i = 1;
(this int n) {
static void Main(string[] args) {
string s = i.ToPrice();
```



# 扩展磁条

你的任务是组织这些磁条，生成以下输出：

**a buck begets more bucks**

Upside命名空间包含扩展方法。Sideways命名空间包含入口点。

Margin类通过增加一个SendIt()方法扩展了string，这个方法只是向控制台写出串，另外还增加一个ToPrice()方法来扩展int，如果int等于1，这个方法返回“a buck”，否则返回“more bucks”。

```
namespace Upside {

public static class Margin {

public static void SendIt
(this string s) {
Console.Write(s);
}

public static string ToPrice
(this int n) {
if (n == 1)
    return "a buck ";
else
    return " more bucks";
}

public static string Green
(this bool b) {
if (b == true)
    return "be";
else
    return "gets";
}
}
}
```

Green方法扩展了bool，如果bool为true，这个方法返回串“be”，如果bool值为false，方法返回“gets”。

在这里Margin类通过增加一个Green()方法扩展了bool。如果bool为true，Green返回“be”，否则返回“gets”。

入口点方法使用了在Margin类中增加的扩展方法。

```
using Upside;
namespace Sideways {

class Program {

static void Main(string[] args) {

int i = 1;
string s = i.ToPrice();
s.SendIt();
bool b = true;
b.Green().SendIt();
b = false;
b.Green().SendIt();
i = 3;
i.ToPrice()
.SendIt();
Console.ReadKey();
}
}
}
```









# TheUNIVERSE

## CAPTAIN AMAZING重生

### 死亡并不是终结!

作者：Bucky Barnes

UNIVERSE STAFF WRITER

OBJECTVILLE

**Captain Amazing实现了逆串行化，傲然归来。**

让人瞠目结舌的转机出现了，Captain Amazing又回到了Objectville。上个月，人们发现Captain Amazing的棺材是空的，里面只留下一个奇怪的便条，他的尸体却不翼而飞。经过对便条的详细分析，查出Captain Amazing对象的DNA，他的所有字段和值都以二进制格式准确无误地保留下来。

今天，这些数据重新焕发生机。Captain又回来了，从他自己绝妙的便条完成了逆串行化。当人们问起他怎么想到这么巧妙的计划时，Captain只是耸耸肩，小声说了句“第9章”。有关消息人士拒绝对Captain这个难解的回答做任何评论，不过确实承认在对Swindler那次失败的追捕之前，Captain花了很多时间看书，研究Dispose方法和持久性。我们期望Captain Amazing……

Captain Amazing 回来了!

……详见A-5版的AMAZING

# 池塘谜题答案



**创建一个新Faucet对象时的输出：**

**back in 20 minutes**

```
public class Faucet {
    public Faucet() {
        Table wine = new Table();
        Hinge book = new Hinge();
        wine.Set(book);
        book.Set(wine);
        wine.Lamp(10);
        book.garden.Lamp("back in");
        book.bulb *= 2;
        wine.Lamp("minutes");
        wine.Lamp(book);
    }
}
```

这就说明了为什么Table必须是一个struct。如果这是一个类，wine就会指向book所指向的同一个对象，这就会导致它覆盖"back in"串。

**附加题：圈出发生装箱的代码。**

由于Lamp()方法取一个对象参数，所以传入一个int或string时会自动发生装箱。

```
public struct Table {
    public string stairs;
    public Hinge floor;
    public void Set(Hinge b) {
        floor = b;
    }
    public void Lamp(object oil) {
        if (oil is int)
            floor.bulb = (int)oil;
        else if (oil is string)
            stairs = (string)oil;
        else if (oil is Hinge) {
            Hinge vine = oil as Hinge;
            Console.WriteLine(vine.Table()
             + " " + floor.bulb + " " + stairs);
        }
    }
}
```

Lamp()方法设置各个string和int。如果对一个int调用这个方法，会设置Hinge所指向对象中的Bulb字段。

如果向Lamp传入一个string，它会将Stairs字段设置为这个串中的内容。

记住，as关键字只用于类，不能用于struct。

```
public class Hinge {
    public int bulb;
    public Table garden;
    public void Set(Table a) {
        garden = a;
    }
    public string Table() {
        return garden.stairs;
    }
}
```

Hinge和Table都有一个Set()方法。Hinge的Set()设置其Table字段Garden，而Table的Set()方法设置其Hinge字段Floor。

# 15 LINQ

# 让数据在你的掌控之中




**这是一个数据驱动的世界…… 最好知道如何在这个世界上生存。**

也许原先你可以编写几天甚至几个星期的程序而不需要处理大量数据，这样的日子已经一去不复返了。如今，一切都离不开数据。实际上，往往不只需要处理一处的数据……另外通常还会采用多种格式。数据库、XML、其他程序中的集合……这些都是一个优秀C#程序员需要面对的任务。这里就可以用到LINQ。LINQ不仅允许你采用一种简单直观的方式查询数据，还可以对数据分组，以及合并不同数据来源的数据。

# 一个简单的项目……

Objectville纸业公司想要同Starbuzz Coffee合作推出一项推广活动。Starbuzz有一个使用很频繁的客户程序，利用这个程序，可以知道谁买了哪种饮料，以及多久买一次。Objectville纸业公司想确定自己的客户中哪些同时是Starbuzz的常客，为他们送出免费咖啡杯，以及他们最喜欢的咖啡的优惠券…… 现在要由你来组合这些数据，生成一个客户名单，为这些客户送出杯子和优惠券。



# ……但是到处都是数据

Starbuzz把所有数据保存在类中，并分组放在一个很大的List中。但是Objectville的数据放在数据库中（可以回顾第1章）。我们希望找出消费超过$90的Starbuzz顾客，将这些顾客与Objectville纸业公司的联系名单匹配，得到最终的名单：我们想得到每个人的名字，他们任职的公司，以及他们最喜欢的Starbuzz饮品。

**Starbuzz数据放在一个List<T>中。**

Starbuzz员工提供了一个程序，这个程序连接到他们的网站，并把所有数据放在一个List<StarbuzzData>中。




需要得到Starbuzz数据的列表，找出与Objectville客户匹配的顾客。

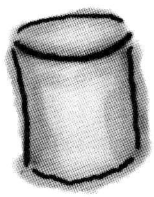
这是Starbuzz代码中的类和enum。

```
class StarbuzzData
{
   public string Name { get; set; }
   public Drink FavoriteDrink { get; set; }
   public int MoneySpent { get; set; }
   public int Visits { get; set; }
}


enum Drink {
   BoringCoffee,
   ChocoRockoLatte,
   TripleEspresso,
   ZestyLemonChai,
   DoubleCappuccino,
   HalfCafAmericano,
   ChocoMacchiato,
   BananaSplitInACup,
}
```

**已经有客户数据。**

第1章构建了Objectville纸业公司的联系表，其中有你需要的部分数据。



ContactDB
数据库

所有Objectville客户数据都放在一个数据库中。

**BRAIN POWER**

怎样结合Starbuzz和Objectville纸业公司的数据，得到一个完整的联系名单呢？

# LINQ可以从多个来源取出数据

LINQ来援助！蜂巢模拟系统中已经使用了LINQ（语言集成查询，Language Integrated Query）来跟踪各组蜜蜂在做什么。在那个程序里，我们利用了LINQ的强大功能编写了一些简单的查询，从一个集合中取出数据。就像处理蜜蜂数据一样，LINQ还可以处理Starbuzz数据，帮助你使用查询取出顾客数据。只要一个集合实现了`IEnumerable<T>`接口，就可以使用LINQ查询。

第12章已经给出LINQ查询的现成代码。后面几页将介绍这个代码如何工作。

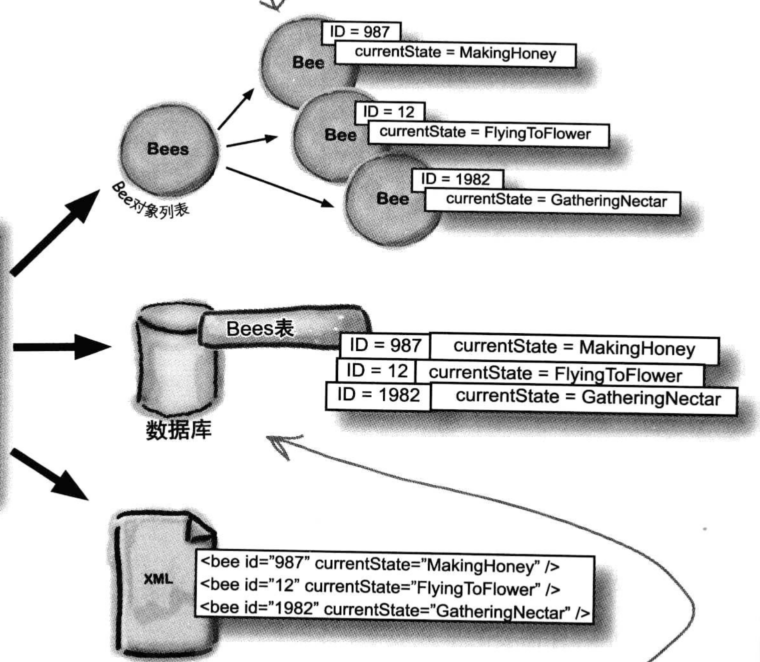
不过LINQ不只是允许你处理集合。还可以使用同样的查询从数据库甚至XML文档中取出数据。所以，既然能够控制集合，同样可以对Objectville数据库使用LINQ。

在这个模拟系统中，蜜蜂存储在一个集合中。

这里是蜜蜂模拟系统中按蜜蜂状态对蜜蜂分组和排序时所用的查询。

```
var beeGroups =
  from bee in world.Bees
  group bee by bee.CurrentState
  into beeGroup
  orderby beeGroup.Key
  select beeGroup;
```

LINQ




需要一个类似的查询取出Starbuzz顾客数据，这些顾客数据也放在一个集合里。

LINQ的一大好处是，对于蜜蜂、顾客或任何其他数据，不论是在数据库还是在XML文档中，都可以使用同样的查询。

LINQ可以处理.NET中使用的几乎每一种数据源。需要在代码文件前面放一个`using System.Linq;`，但是仅此而已。更棒的是，IDE会在创建的所有代码文件前面自动增加LINQ的一个引用。

# 已经为LINQ建立了.NET集合

在第8章我们已经了解到，.NET中的所有集合类型都实现了IEnumerable<T>接口。不过现在花点时间复习一下：在IDE窗口中键入System.Collections.Generic.IEnumerable<int>，右键点击这一行，选择 Go To Definition（或按下F12）。可以看到IEnumerable接口定义了一个GetEnumerator()方法：

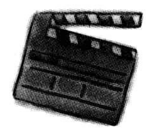
注意IEnumerable<T>扩展了一个名为IEnumerable的接口。也可以使用“Go to Definition”来深入研究。

```
namespace System.Collections.Generic {
  interface IEnumerable<T> : IEnumerable {
    // Summary:
    //     Returns an enumerator that iterates through the collection.
    //
    // Returns:
    //     A System.Collections.Generic.IEnumerator<T> that can be
    //       used to iterate through the collection.
    IEnumerator<T> GetEnumerator();
  }
}
```

这是这个接口中唯一的方法。每个集合都要实现这个方法。也可以创建你自己的对象类型实现IEnumerable<T>接口……如果实现了这个接口，就可以对这个对象使用LINQ了。

这个方法要求对象定义一种遍历元素的方式，一次处理一个元素。这也是使用LINQ唯一的前提要求。如果可以逐项地遍历一个数据列表，就能实现IEnumerable<T>，LINQ就能查询这个集合。

## 在幕后



LINQ使用扩展方法允许查询、排序和更新数据。可以自己试试看。创建一个名为linqtest的int数组，在这个数组中放一些数，然后键入下面这行代码（不必担心，稍后就会了解这行代码是什么意思）：

```
IEnumerable<int> result = from i in linqtest where i < 3 select i;
```

下面把你创建的文件前面的using System.Linq;行注释掉。重新构建这个解决方案，会看到这一行无法通过编译。使用LINQ调用的方法实际上是用来扩展数组的一些扩展方法。

现在可以看出第14章的扩展方法为什么如此重要……利用扩展方法，.NET（和你）就能为现有的类型增加各种各样绝妙的行为。

# 利用LINQ可以轻松地查询

以下是LINQ语法的一个简单例子。它选择一个int数组中所有小于37的数，并按升序排列这些数。这里使用了4个子句来完成查询，分别告诉它查询哪个集合，使用什么条件来确定选择集合中的哪些成员，如何对结果排序，以及结果应当如何返回。

```
int[] values = new int[] {0, 12, 44, 36, 92, 54, 13, 8};

var result = from v in values

             where v < 37

             orderby v

             select v;

foreach(int i in result)

  Console.Write("{0} ", i);

Console.ReadKey();
```

这里指定"v"代表查询中values数组的各个值。所以v将是0，然后是12，再然后是44，36……。这称为范围变量。

这个LINQ查询有4个子句：from子句，where子句，orderby子句和select子句。

这表示，选择数组中小于37的各个v。

然后，按顺序放置这些值（从低到高）。

如果以前用过SQL，会发现把select放在最后看起来很奇怪，不过LINQ中就是这么做的。

现在可以迭代处理LINQ返回的序列来打印输出。

**输出：**

0 8 12 13 36

## var

var是一个关键字，告诉编译器在编译时确定一个变量的类型。.NET根据使用LINQ查询的局部变量的类型来检测这个类型。构建解决方案时，编译器会把var替换为处理的数据的正确类型。

在前面的例子中，编译这行代码时：

```
var result = from v in values
```

编译会把"var"替换为：

```
IEnumerable<int>
```

讨论集合的接口时，要记住IEnumerable<T>是支持迭代的接口。很多非凡的LINQ查询都是使用扩展IEnumerable<T>的扩展方法实现的，所以你会大量看到这个接口。



翻回到第8章，复习一下IEnumerable<T>接口。另外，在附录"其他"的第6项中可以有更多了解。

# LINQ很简单，但是你的查询不一定简单

Jimmy想把他的小公司卖给一家大投资商，希望赚些钱买他能找到的最贵Captain Amazing漫画。如何使用LINQ帮助他处理两个集合来确定哪些漫画是最贵的？

**1** Jimmy从Captain Amazing漫画迷页面下载了各期Captain Amazing漫画的一个列表。他把这些数据放在一个Comic对象List<T> 中，Comic对象有两个字段：Name和Issue。

```
class Comic {
    public string Name { get; set; }
    public int Issue { get; set; }
}
```

Jimmy使用对象初始化方法和一个集合初始化方法来建立他的目录：

```
private static IEnumerable<Comic> BuildCatalog()
{
    return new List<Comic> {
        new Comic { Name = “Johnny America vs. the Pinko”, Issue = 6 },
        new Comic { Name = “Rock and Roll (limited edition)”, Issue = 19 },
        new Comic { Name = “Woman’s Work”, Issue = 36 },
        new Comic { Name = “Hippie Madness (misprinted)”, Issue = 57 },
        new Comic { Name = “Revenge of the New Wave Freak (damaged)”, Issue = 68 },
        new Comic { Name = “Black Monday”, Issue = 74 },
        new Comic { Name = “Tribal Tattoo Madness”, Issue = 83 },
        new Comic { Name = “The Death of an Object”, Issue = 97 },
    };
}
```

这个方法是静态的，不过这并没有什么特殊的原因，只不过是希望能够容易地从一个控制台应用的入口点方法调用。

这里从集合和对象初始化方法去掉了<Comic>后面的()，因为并不需要这个括号。

#74期Captain Amazing名为“Black Monday”

花点时间翻到附录“其他”中的第6项，稍稍了解一下语法，这会很有帮助。可以利用这个好机会试验一下！

**2** 幸运的是，Greg’s List网站上Captain Amazing漫画正在热卖。他知道#57期“Hippie Madness”有印刷错误，这一期的几乎所有漫画都印错了，Jimmy发现Greg’s List最近有一个珍本卖到$13525。经过几个小时的搜索，Jimmy终于构建了一个Dictionary<>，可以将漫画的期号映射到价格。

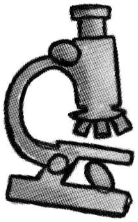
```
private static Dictionary<int, decimal> GetPrices()
{
    return new Dictionary<int, decimal> {
        { 6, 3600M },
        { 19, 500M },
        { 36, 650M },
        { 57, 13525M },
        { 68, 250M },
        { 74, 75M },
        { 83, 25.75M },
        { 97, 35.25M },
    };
}
```

还记得第8章中字典集合初始化方法的这个语法吧？

#57期价值$13,525。

**BRAIN POWER**

仔细看690页上的LINQ查询。你认为Jim要找出最贵的那期漫画需要写一个怎样的查询？



# 查询剖析

Jimmy用一个LINQ查询就可以分析他的漫画数据。where子句告诉LINQ集合中的哪些项应该包括在结果中。但是这个子句不一定是一个简单的比较。其中可以包含任何合法的C#表达式—例如，使用values字典来告诉它只返回价格超过$500的漫画。另外orderby子句也类似，可以告诉LINQ按价格对漫画排序。

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();
```

LINQ查询从漫画列表取出Comic对象，这里使用了values字典中的值来确定选择哪些漫画。

```
var mostExpensive =
    from comic in comics
    where values[comic.Issue] > 500
    orderby values[comic.Issue] descending
    select comic;
```

查询中的第一个子句是from子句。这个子句告诉LINQ查询comics集合，comic将在这个查询中用于指示如何处理集合中的各部分数据。

使用from子句时可以选择你喜欢的任何名字，我们使用了"comic"。

where和orderby子句可以包含任何C#语句，所以可以使用values字典来选择价格超过$500的漫画，并且可以对结果排序，使最贵的漫画放在最前面。

在from子句中特别定义了名comic，所以可以在where和orderby子句中使用。

```
foreach (Comic comic in mostExpensive)
    Console.WriteLine("{0} is worth {1:c}",
                      comic.Name, values[comic.Issue]);
```

向WriteLine输出增加"{1:c}"时，就是在告诉它采用本地货币格式打印第二个参数。

这个查询将其结果返回到一个名为mostExpensive的IEnumerable<T>中。select子句确定哪些要放在结果中，因为这里选择了comic，所以查询会返回Comic对象。

**输出:**

```
Hippie Madness (misprinted) is worth $13,525.00
Johnny America vs. the Pinko is worth $3,600.00
Woman's Work is worth $650.00
```

如果你没有用过SQL也不用担心，使用LINQ不要求你掌握SQL的知识。但是如果你很好奇，可以看看“Head First SQL”。




**LINQ看起来可能很像SQL，但是它的工作与SQL并不一样**

如果以前经常使用SQL，可能会认为这些LINQ查询很浅显，很简单，像你这样想的人并不少，因为很多开发人员都犯过这个错误。确实，LINQ使用了select, from, where, descending和join关键字，这些都是从SQL借用的。但是LINQ与SQL完全不同，如果你把LINQ当成SQL来理解，最后得到的代码可能根本无法完成你原来期望实现的任务。

这二者之间的一个主要区别在于，SQL处理数据库表，而数据库表与可枚举的对象有很大不同。一个非常重要的区别是SQL表没有顺序，而可枚举的对象是有顺序的。对一个数据库表执行一个SQL查询时，可以相信这个表不会更新。SQL内置了各种可以信赖的数据安全性。

如果你想了解究竟，需要知道：SQL查询是集（set）操作，也就是说，它们不会按预期的顺序检查表中的行。而另一方面，集合可以存储任何内容，可以是值、struct、对象，或者是其他数据，而且集合有一个特定的顺序（数据库表中的行没有特定的顺序，除非建立一个SQL查询来排序，而集合与此不同，List中的项是有顺序的）。LINQ允许对集合完成所支持的任何操作，甚至可以对集合中的对象调用方法。LINQ会循环处理集合，这说明，它会以一种特定的顺序完成操作。这一点看起来好像不太重要，但是如果你习惯于使用SQL，这意味着倘若你以为LINQ查询会表现得像SQL查询，可能会失望。

LINQ和SQL之间还有其他一些不同。不过要想现在开始使用LINQ，并不需要深入了解所有这些区别！只要以一种开放的心态来看待LINQ，不要期望它与SQL的做法相同。

# LINQ是个多面手

LINQ并不只是能够从一个集合中取出一些项。在返回这些项之前还可以对它们进行修改。生成一组结果序列后，LINQ提供了大量方法来处理这些结果。总之，LINQ提供了管理数据所需的各种工具。

**所有集合都是可枚举的，它们实现了IEnumerable<T>，不过理论上讲并不是所有可枚举的对象都是集合，除非它实现了ICollection<T>接口，这说明要实现Add()、Clear()、Contains()、CopyTo()和Remove()……当然，ICollection<T>扩展了IEnumerable<T>。LINQ处理的是值或者对象的序列，而不是集合。要得到一个序列，只需一个实现了IEnumerable<T>的对象。**

**修改查询返回的各个项。**

这个代码会在数组中每个串的最后追加一个串。它不会改变数组本身—这里将创建一个新序列，其中包含修改后的串。

```
string[] sandwiches = { "ham and cheese", "salami with mayo",
                        "turkey and swiss", "chicken cutlet" };
var sandwichesOnRye =
    from sandwich in sandwiches
    select sandwich + " on rye";

foreach (var sandwich in sandwichesOnRye)
    Console.WriteLine(sandwich);
```

这会把串"on rye"追加到查询返回的结果数组中的每一项。

注意返回的所有项最后都追加了"on rye"。

这里对查询结果中的各项做了修改……不过并没有改变原集合或数据库中的项。

**输出:**

```
ham and cheese on rye
salami with mayo on rye
turkey and swiss on rye
chicken cutlet on rye
```

**对集合完成计算。**

要记住，我们说过LINQ向集合（和数据库访问对象，以及任何实现了IEnumerable<T>的对象）提供了扩展方法……其中一些方法本身就很有用，甚至不需要查询：

```
Random random = new Random();
List<int> listOfNumbers = new List<int>();
int length = random.Next(50, 150);
for (int i = 0; i < length; i++)
    listOfNumbers.Add(random.Next(100));

Console.WriteLine("There are {0} numbers",
                  listOfNumbers.Count());
Console.WriteLine("The smallest is {0}",
                  listOfNumbers.Min());
Console.WriteLine("The biggest is {0}",
                  listOfNumbers.Max());
Console.WriteLine("The sum is {0}",
                  listOfNumbers.Sum());
Console.WriteLine("The average is {0:F2}",
                  listOfNumbers.Average());
```

所有这些方法都不是.NET集合类的方法……它们都由LINQ定义。

这些都是System.Linq命名空间中使用静态类Enumerable为Enumerable<T>定义的扩展方法。不过不要光听我们说！点击其中任何一个方法，使用"Go to Definition"自己试试看。

序列是一组有序的对象或值，LINQ在IEnumerable<T>中返回的就是序列。

**将所有或部分结果保存在一个新序列中。**
有时你可能希望保存一个LINQ查询的结果。可以使用ToList()命令来做到：

```
var under50sorted =
  from number in listOfNumbers
  where number < 50
  orderby number descending
  select number;

List<int> newList = under50sorted.ToList();
```

这一次将一个数字列表以降序排序，即从高到低排列。

ToList()将一个LINQ var转换为一个List<T>对象，来保留查询的结果。另外还有ToArray()和ToDictionary()方法，它们的作用从名字就可以看出来。

甚至可以使用Take()方法只取结果的一个子集：

```
var firstFive = under50sorted.Take(6);

List<int> shortList = firstFive.ToList();
foreach (int n in shortList)
    Console.WriteLine(n);
```

Take()从一个LINQ查询的结果中根据指定的数目取出前面的部分项。可以把这些项放在另一个var中，然后再把它转换到一个列表中。




**访问LINQ查询的结果之前并不会真正运行LINQ查询！**

这称为“延迟计算”—在执行一个使用了查询结果的语句之前，LINQ查询不会真正完成任何循环处理。正因如此ToList()很重要：它会告诉LINQ立即执行查询。

**查看Microsoft官方的“101 LINQ Samples”网页。**
LINQ能做的工作还有很多。幸运的是，Microsoft提供了一个很棒的参考可以提供帮助。

**http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx**

## there are no Dumb Questions

**问：这里有很多新的关键字：from, where, orderby, select…… 就像一个完全不同的语言。为什么这看起来与C#的其他语法差别这么大？**

**答：** 因为它的作用不同。大多数C#语法都设计为一次完成一个小操作或计算。可以开始一个循环，或者设置一个变量，完成一个数学运算，调用一个方法…… 所有这些都是单个操作。LINQ查询看起来有所不同，是因为一个LINQ查询通常一次会完成很多工作。下面更仔细地分析一个简单的查询：

```
var under10 =
 from number in numberArray
 where number < 10
 select number;
```

看起来很简单，这里没有太多内容，是不是？但实际上这是一个相当复杂的代码。考虑一下，要让程序从numberArray中选择所有小于10的数要做些什么。首先，需要循环处理整个数组。然后，将每个数与10进行比较。再收集这些结果以便代码使用。

正是因为这一点，LINQ看起来有些奇怪：因为C#必须把大量行为压缩在几行代码里。

**利用LINQ只需使用很少的代码就可以编写完成复杂工作的查询。**

## BULLET POINTS

- **from**子句用于指定所查询的IEnumerable<T>。后面总跟有一个变量名，再后面是in以及输入名（(from value in values)。
- **where**通常跟在from子句后面。在这个子句中可以使用正常的C#条件来告诉LINQ从集合中取出哪些项(where value < 10)。
- 利用**orderby**可以对结果排序。它后面是用来完成排序的条件，还可以跟有**descending**（可选）指定按降序排序(orderby value descending)。
- **select**子句用于指定将什么放在结果中(select value)。
- 利用**Take**可以从LINQ查询的结果中取出前面几项(results.Take(10))。LINQ还为集合提供了其他一些方法：Min()，Max()，Sum()和Average()。
- 可以用select选择任何内容，而不仅限于选择from子句中创建的名。下面是一个例子：如果LINQ查询从一个int值数组中取出一组价格，并在from子句中命名为value，可以如下返回一个价格串序列：select String.Format("{0:c}", value)。

这很类似于第9章中构建十六进制转储工具时使用的{0:x}。还有其它一些格式，{0:d}和{0:D}对应短日期和长日期，{0:P}或{0:Pn}用于打印一个百分数（有n位小数）。

## there are no Dumb Questions

**问：** from子句是怎样工作的？

**答：** 这非常类似于foreach循环的第一行。LINQ查询之所以有点困难，原因就在于这里不只是完成一个操作。

LINQ查询对一个集合中的各项反复做同一件事。from子句会完成两个工作：它告诉LINQ查询使用哪个集合，另外为所查询集合中的各个成员指定一个名。

from子句为集合中的各个项创建一个新的名，这一点非常类似于foreach循环的做法。以下是一个foreach循环的第一行：

```
foreach (int i in values)
```

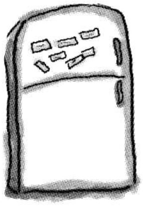
这个foreach循环临时创建一个变量i，它会顺序地赋为values集合中的各个项。现在来看同一个集合的LINQ查询中的from子句：

```
from i in values
```

这个子句看起来非常相似。它创建一个临时变量i，并顺序地赋为values集合中的各个项。foreach循环对集合中的每一项运行同样的代码块，而LINQ查询对集合中的每一项应用where子句中同样的规则，来确定是否把这一项包括在结果中。不过这里有一点要记住，LINQ查询只是一些扩展方法。它们会调用真正完成工作的方法。你也可以直接调用这些方法而不使用LINQ。

**问：** LINQ如何确定哪些内容要放在结果中？

**答：** 这正是select子句的作用。每个LINQ 查询都会返回一个序列，这个序列中的每一项都有相同的类型。select子句告诉LINQ这个序列中应该有什么。查询某个类型的数组或列表时（如一个int数组或List<string>），select子句中放什么是很显然的。但是，如果从一个Comic对象列表中选择呢？可以像Jimmy一样，选择整个类。不过也可以把查询的最后一行修改为select comic.Name，告诉它返回一个string序列。或者可以替换为select comic.Issue，让它返回一个int序列。



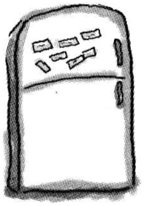
## LINQ磁条

重新组织磁条，生成下面的输出。

```
pigeon descending
Console.WriteLine("Get your kicks on route {0}",
weasels.Sum()
sparrow in bears
pigeon in badgers
where
from
from
orderby
select
select
var weasels =
int[] badgers =
var skunks =
var bears =
);
skunks
.Take(3);
pigeon + 5;
sparrow - 1;
(pigeon != 36 && pigeon < 50)
{ 36, 5, 91, 3, 41, 69, 8 };
```

**输出:**

**Get your kicks on route 66**



# LINQ磁条答案

重新组织磁条，生成下面的输出。

首先，要为LINQ提供某种序列、集合或数组，在这里是一个整数数组。

```
int[] badgers = { 36, 5, 91, 3, 41, 69, 8 };
```

“from pigeon in badgers”用来出题很好，但作为一个LINQ查询可读性不好，如果是“from badger in badgers”会更可读。

```
var skunks =
    from pigeon in badgers
    where (pigeon != 36 && pigeon < 50)
    orderby pigeon descending
    select pigeon + 5;
```

执行这条语句后，skunks包含4个数：46，13，10和8。

这个LINQ语句取出数组中所有小于50而且不等于36的数，对它们分别加5，按从大到小顺序排序，再把它们放在一个新对象中，并让skunks引用指向这个对象。

```
var bears = skunks.Take(3);
```

执行这条语句后，bears中包含3个数：46，13和10。

这里取出skunks中的前3个数，并把它们放在一个名为bears的新序列中。

```
var weasels =
    from sparrow in bears
    select sparrow - 1;
```

执行这条语句之后，weasels包含3个数：45，12和9。

这条语句只是将bears中的各个数分别减1，并把它们放在weasels中。

```
Console.WriteLine("Get your kicks on route {0}",
    weasels.Sum() );
```

$45 + 12 + 9 = 66$

weasels中的数累加得到66。

**输出：**

**Get your kicks on route 66**

# LINQ可以将结果合并分组

你已经知道可以使用LINQ将结果分组，因为我们在蜂巢模拟系统中就曾经这样做过。下面再仔细分析这个查询，看看它是怎么做的。

```
var beeGroups =

  from bee in world.Bees

  group bee by bee.CurrentState

  into beeGroup

  orderby beeGroup.Key

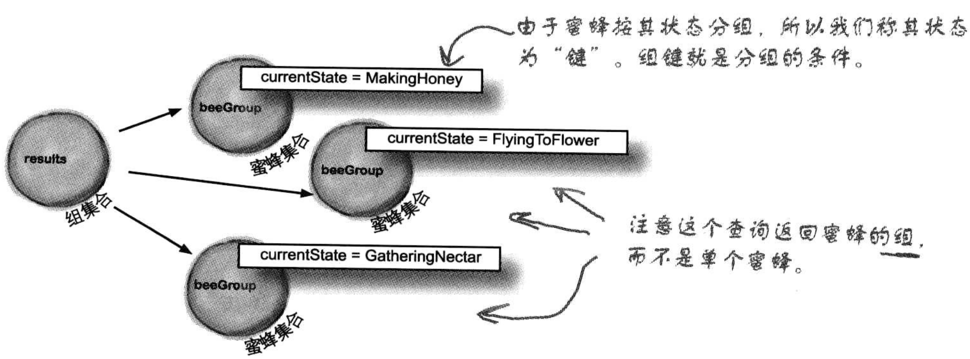
  select beeGroup;
```

**1** 这个查询的开头与之前见到的其他查询是一样的，从`world.Bees`集合（这是一个`List<Bee>`对象）取出单个蜜蜂对象。

**2** 这个查询中的下一行有一个新关键字：**`group`**。这就告诉查询要返回蜜蜂的组。这表示，不是返回一个序列，而要返回一个序列的序列。**`group bee by bee.CurrentState`**告诉LINQ：在所选择蜜蜂中，将不同`CurrentState`属性的蜜蜂分组返回。最后，需要向LINQ为这个组提供一个名。这就是下一行的作用：**`into beeGroup`**指出名“beeGroup”表示这些新的组。

**3** 既然得到了这些组，下面可以对它们进行处理。由于我们要返回一个组序列，可以使用`orderby`关键字将这些组按`CurrentState enum`值的顺序排列（`Idle`，`FlyingToFlower`等）：**`orderby beeGroup.Key`**告诉这个查询将组序列按顺序排列，即按组键排序。由于这里按蜜蜂的`CurrentState`对蜜蜂分组，所以这就会用作为组键。

**4** 现在只需使用`select`关键字指示查询返回什么结果。由于我们要返回组，所以选择组名：**`select beeGroup;`**



# 将Jimmy漫画书的价格合并分组

Jimmy买了很多便宜的漫画书，一些价格中等的漫画书，还有一些很贵的书，他希望在决定买哪些漫画书之前先要知道有哪些选择。他从Greg's List得到了三种价格，使用他的GetPrices()方法把这些价格放入一个Dictionary<int,int>，下面使用LINQ将其分为三组：一组是价格在$100以下的便宜漫画书，另一组是价格在$100和$1000之间的中等漫画书，还有一组是价格超过$1000的昂贵的漫画书。我们将创建一个PriceRange enum用作为这些组的键，还会创建一个EvaluatePrice()方法来分析价格并返回一个PriceRange。

1. **每个组需要一个键，我们将使用一个enum。**
组键是这个组中所有成员共同的东西。键可以是任何类型，可以是一个串，一个数，甚至可以是一个对象引用。我们将查看Jimmy从Greg's list得到的所有价格。查询返回的每一组是一个期号序列，组键是一个PriceRange enum。EvaluatePrice()方法取一个价格作为参数，并返回一个PriceRange：

```
enum PriceRange { Cheap, Midrange, Expensive }

static PriceRange EvaluatePrice(decimal price) {
    if (price < 100M) return PriceRange.Cheap;
    else if (price < 1000M) return PriceRange.Midrange;
    else return PriceRange.Expensive;
}
```

2. **现在可以按价格类别对漫画分组。**
LINQ查询返回一个序列的序列。结果中的每个序列有一个Key属性，它与EvaluatePrice()方法返回的PriceRange匹配。仔细查看group by子句，这里从Dictionary取出键值对，并使用pair表示每一个键值对：pair.Key是期号，pair.Value是从Greg's list得到的价格。group pair.Key告诉LINQ要创建期号的组，然后根据EvaluatePrice()方法返回的价格类别进行分组：

```
Dictionary<int, decimal> values = GetPrices();

var priceGroups =
    from pair in values
    group pair.Key by EvaluatePrice(pair.Value)
        into priceGroup
        orderby priceGroup.Key descending
        select priceGroup;

foreach (var group in priceGroups) {
    Console.Write("I found {0} {1} comics: issues ", group.Count(), group.Key);
    foreach (var price in group)
        Console.Write(price.ToString() + " ");
    Console.WriteLine();
}
```

这个查询将价格发送给EvaluatePrice()来得出一个特定价格属于哪一组。这会返回一个PriceRange enum，它将用作为组键。

每一组都是一个序列，所以我们增加了一个内层foreach循环，从组中取出各个价格。

**输出：**

```
I found 2 Expensive comics: issues 6 57
I found 3 Midrange comics: issues 19 36 68
I found 3 Cheap comics: issues 74 83 97
```

# 池塘谜题



你的任务是从池塘里取出代码片段，在程序在填空。一个代码片段可以使用多次，另外不是所有代码片段都会用到。你的目标是建立一个能生成以下输出的代码：

**Horses enjoy eating carrots, but they love eating apples.**

```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}
```

提示：LINQ按字母顺序对串排序。

```
Line[] lines = {
  new Line( new string[] { "eating", "carrots,",
          "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay",
         "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!",
         "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they",
         "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 ) };
```

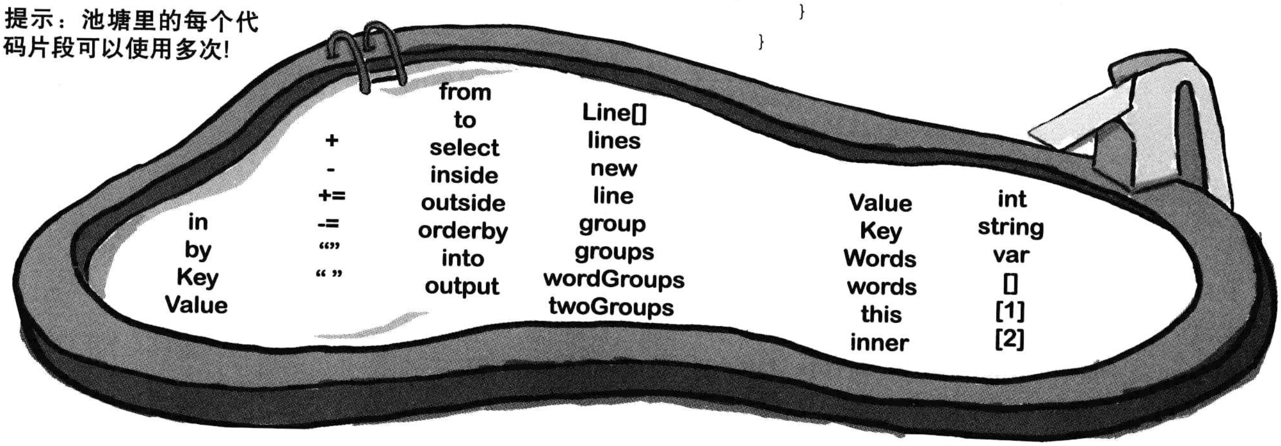
```
var _______ =
    from _______ in _______
    _______ line by line._______
    into wordGroups
    orderby _________._______
    select _________;

_____ __________ = words._______(2);

foreach (var group in twoGroups)
{
   int i = 0;

   foreach (_______ inner in _______) {
      i++;
      if (i == _______.Key) {
         var poem =
            _______ word in _______._______
            _________ word descending
            _______ word + ____;
         foreach (var word in _______)
            Console.Write(word);
      }
   }
}
```

提示：池塘里的每个代码片段可以使用多次！



# 池塘谜题答案



```
class Line {
    public string[] Words;
    public int Value;
    public Line(string[] Words, int Value) {
        this.Words = Words; this.Value = Value;
    }
}

Line[] lines = {
  new Line( new string[] { "eating", "carrots,", "but", "enjoy", "Horses" } , 1),
  new Line( new string[] { "zebras?", "hay", "Cows", "bridge.", "bolted" } , 2),
  new Line( new string[] { "fork", "dogs!", "Engine", "and" }, 3 ) ,
  new Line( new string[] { "love", "they", "apples.", "eating" }, 2 ) ,
  new Line( new string[] { "whistled.", "Bump" }, 1 )
};

var words =
    from line in lines
    group line by line.Value
    into wordGroups
    orderby wordGroups.Key
    select wordGroups;

var twoGroups = words.Take(2);

foreach (var group in twoGroups)
{
    int i = 0;
    foreach (var inner in group) {
        i++;
        if (i == group.Key) {
            var poem =
                from word in inner.Words
                orderby word descending
                select word + " ";
            foreach (var word in poem)
                Console.Write(word);
        }
    }
}
```

第一个LINQ查询将lines[]中的Line对象分为两组，按其Value分组，并按Value键的升序排列。

前两个组是Value为1和2的Line。

这个循环对第一组的第一个Line对象和第二组的第二个Line对象完成一个LINQ查询。

"Horses enjoy eating carrots, but" 和 "they love eating apples" 是按字母降序排列，你发现了吗？

**输出：** **Horses enjoy eating carrots, but they love eating apples.**

# 使用join将两个集合合并到一个查询

Jimmy买了大量漫画书，构成了一个很大的集合，他想与从Greg’s List得到的价格做个比较，看看他买的是不是划算。他一直使用一个`Purchase`类来跟踪他购买的漫画，这个类包含两个字段：`Issue`和`Price`。Jimmy有一个`List<Purchase>`，名为`purchases`，其中包含他购买的所有漫画书。不过，现在需要把他买的漫画书与Greg’s List上看到的价格进行比较。怎么做呢？

LINQ可以解决问题！利用`join`关键字，可以把两个集合中的数据合并到一个查询中。为此，它将第一个集合中的项与第二个集合中相应的项进行比较(LINQ很聪明，可以很高效地完成这个工作，除非很有必要，否则它并不会对每一对项都进行比较）。最终结果将合并匹配的每一对项。

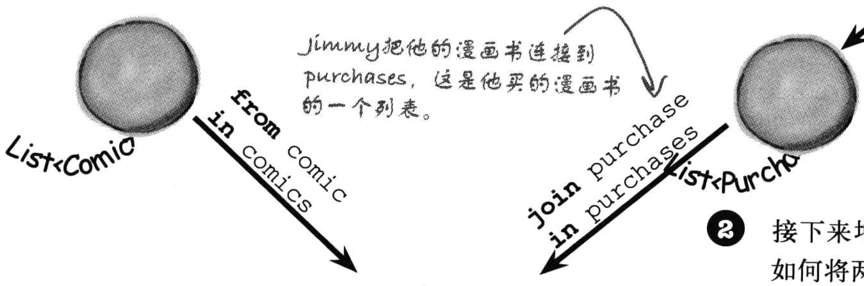
❶ 查询最前面仍然是`from`子句。不过，后面不再是用来确定结果的条件，而是增加了：

**join** *name* **in** *collection*

这个`join`子句告诉LINQ循环处理两个集合，匹配各个集合中的一对成员。它将每次迭代时从连接的这两个集合中取出的成员命名为`name`。可以在`where`子句中使用这个name。

Jimmy的数据放在一个名为purchases的Purchase对象集合中。

```
class Purchase {
  public int Issue
             { get; set; }
  public decimal Price
             { get; set; }
}
```



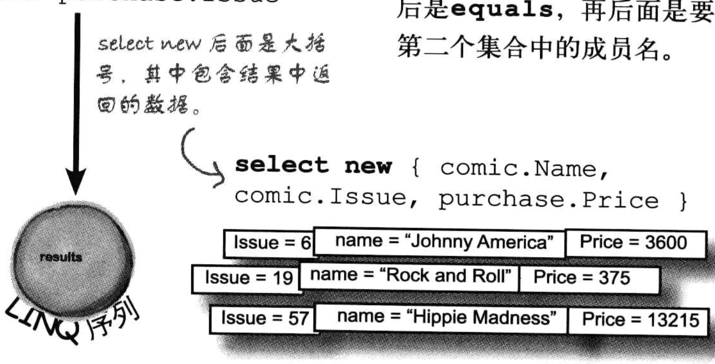

```
on comic.Issue
equals purchase.Issue
```

❷ 接下来增加on子句，这会告诉LINQ如何将两个集合匹配在一起。后面是所匹配的第一个集合中的成员名，然后是**equals**，再后面是要匹配到的第二个集合中的成员名。

❸ `where`和`orderby`子句还是与以往LINQ查询中一样。最后可以是一个正常的`select`子句，不过通常希望返回的结果从一个集合取出一些数据，再从另一个集合中取出另外一些数据。所以这里使用**select new**利用一个匿名类型创建一个定制的结果集。

select new后面是大括号，其中包含结果中返回的数据。

```
select new { comic.Name,
comic.Issue, purchase.Price }
```




**翻到附录“其他”中的第8项，了解更多有关匿名类型（anonymous type）的内容！**

# Jimmy省了一大笔钱

看起来Jimmy买得很划算。他创建了一个Purchase类列表，其中包含他买的所有漫画书，并与Greg's List上得到价格进行比较。

**1 首先Jimmy创建要连接的集合。**

Jimmy已经有了第一个集合（只需要使用先前的BuildCatalog()方法）。所以接下来要编写一个FindPurchases()方法来构建他的Purchase类列表。

```
static IEnumerable<Purchase> FindPurchases() {
    List<Purchase> purchases = new List<Purchase>() {
        new Purchase() { Issue = 68, Price = 225M },
        new Purchase() { Issue = 19, Price = 375M },
        new Purchase() { Issue = 6, Price = 3600M },
        new Purchase() { Issue = 57, Price = 13215M },
        new Purchase() { Issue = 36, Price = 660M },
    };
    return purchases;
}
```

Jimmy花了$13,215买下#57期。

**2 现在可以完成连接！**

你已经看到了这个查询的各个部分…… 下面可以把它们集成在一起。

```
IEnumerable<Comic> comics = BuildCatalog();
Dictionary<int, decimal> values = GetPrices();
IEnumerable<Purchase> purchases = FindPurchases();
var results =
    from comic in comics
    join purchase in purchases
    on comic.Issue equals purchase.Issue
    orderby comic.Issue ascending
    select new { comic.Name, comic.Issue, purchase.Price };
decimal gregsListValue = 0;
decimal totalSpent = 0;
foreach (var result in results) {
    gregsListValue += values[result.Issue];
    totalSpent += result.Price;
    Console.WriteLine("Issue #{0} ({1}) bought for {2:c}",
               result.Issue, result.Name, result.Price);
}
Console.WriteLine("I spent {0:c} on comics worth {1:c}",
               totalSpent, gregsListValue);
```

Jimmy使用一个join子句时，LINQ会把comics集合中的第一项与purchases集合中的第一项进行比较，查看哪些项的comic.Issue等于purchase.Issue。

select new子句创建了一个结果集，包含comic成员的Name和Issue，以及purchase成员的Price。

**输出：**

```
Issue #6 (Johnny America vs. the Pinko) bought for $3,600.00
Issue #19 (Rock and Roll (limited edition)) bought for $375.00
Issue #36 (Woman's Work) bought for $660.00
Issue #57 (Hippie Madness (misprinted)) bought for $13,215.00
Issue #68 (Revenge of the New Wave Freak (damaged)) bought for $225.00
I spent $18,075.00 on comics worth $18,525.00
```

Jimmy很高兴自己了解LINQ，因为利用LINQ能看出他省了钱。

那好，现在我知道Jimmy可以使用LINQ查询来查询集合，处理他的漫画书…… 但是前面提到的Starbuzz推广活动呢？我还是不知道怎么利用LINQ处理数据库。



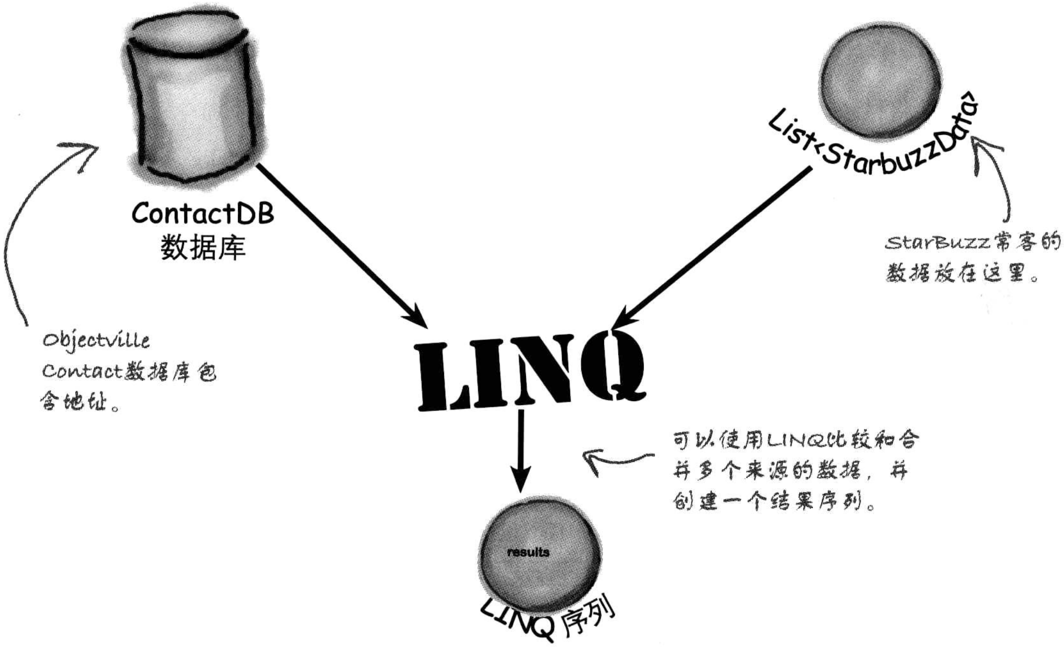
**LINQ处理数据库和处理集合所用的语法完全相同。**

在第1章已经看到，利用.NET处理数据库非常容易。在IDE中，可以非常方便地连接数据库、增加表，甚至可以将这些表中的数据链接到窗体。

尽管LINQ to SQL在底层与LINQ是完全不同的，不过，编写代码时，看起来与其他LINQ查询确实很相似。

现在，还是用之前连接的那个数据库，不过要用LINQ来查询这个数据库。不仅如此，利用LINQ还可以将数据库中的数据与对象中的数据无缝地结合。

实际上，可以使用完全相同的查询语法……只需访问数据库，对它运行LINQ查询。



# 将LINQ连接到一个SQL数据库

LINQ可以处理实现了IEnumerable<T>接口的对象，对不对？所以应该可以使用一个实现了IEnumerable的对象来访问SQL数据库。利用C#，可以很容易地为工程增加这个对象。

动手做

**1 向一个新的控制台应用工程增加Objectville Contact数据库。**

再来看第1章，你为Objectville纸业公司创建了一个联系信息SQL Server Compact数据库，并把它保存在一个名为ContactDB.sdf的文件中。创建一个新的 Windows Application工程，在Solution Explorer中右键点击这个工程，选择“Add Existing Item”，并增加这个数据库。要从文件类型过滤器下拉列表中选择“Data Files”，导航到这个SQL数据库文件，把它增加到工程中（IDE会弹出一个数据源配置向导，不过可以将它取消）。

**2 使用SqlMetal.exe程序生成LINQ to SQL类。**

要连接SQL数据库和你的代码还需要一个步骤，可以使用一个名为**SqlMetal.exe**的程序来完成。这是一个命令行工具，随Visual Studio 2010安装（你安装的是哪个版本并不重要，所有版本都提供了这个工具）。可以在一个名为“Microsoft SDKs”的文件夹中找到这个程序，这个文件夹位于“Program Files”文件夹下（或者，如果你运行的是64位的Windows，则在“Program Files (x86)”文件夹下）。打开一个命令提示窗口，运行一个命令把Microsoft SDKs文件夹增加到你的路径。如果使用64位的Windows，需要键入以下命令：

```
PATH=%PATH%;%ProgramFiles(x86)%\Microsoft SDKs\Windows\v7.0A\Bin\
```

或者如果使用32位的版本，要键入以下命令：

```
PATH=%PATH%;%ProgramFiles%\Microsoft SDKs\Windows\v7.0A\Bin\
```

可能还在Bin\文件夹下一个名为“NETFX 4.0 Tools”的文件夹中。如果是这样，需要把“NETFX 4.0 Tools”增加到PATH=命令的最后。

接下来，将目录切换到你的工程文件夹（cd 文件夹名），并键入以下命令：

```
SqlMetal.exe ContactDB.sdf /dbml:ContactDB.dbml
```

运行这个命令应该能看到以下结果：

```
Microsoft (R) Database Mapping Generator 2008 version 1.00.30729
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

完成这个工作后，你的文件夹中应该包含3个新文件：ContactDB.dbml、ContactDB.designer.cs和ContactDB.dbml.layout。

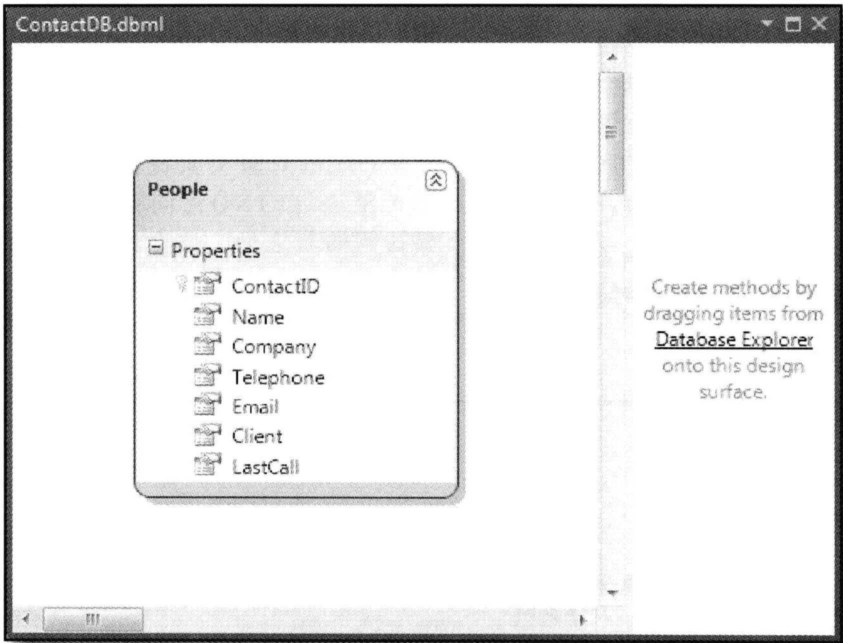
使用“Add Existing Item”将ContactDB.dbml增加到工程（同样地，从文件类型过滤器下拉列表中选择“Data Files”）。增加这个文件时，IDE也会自动增加其他文件。

> 可以在这里更多地了解SqlMetal.exe:
> http://msdn.microsoft.com/en-us/library/bb386987.aspx
> 如果出于某种奇怪的原因Visual Studio安装程序没有为你安装这个工具，可以在这个页面上找到Microsoft SDK下载页面的链接。

**3 在对象关系设计工具（Object Relational Designer）中打开LINQ to SQL类。**

运行`SqlMetal.exe`创建`ContactDB.dbml`和其他文件并把它们增加到工程时，也就创建了LINQ to SQL类。要记住，LINQ查询用来处理实现了`IEnumerable<T>`接口的对象。便是SQL Server Compact数据库根本不是一个对象！这里就要用到LINQ to SQL类。这些类知道如何查询数据库中的表，而且实现了`IEnumerable<T>`，并有一个枚举器可以返回表中的数据。

IDE提供了一个很棒的工具，名为对象关系设计工具，可以准确地显示用`SqlMetal.exe`生成了哪些类。一旦将`ContactDB.dbml`增加到你的工程，双击这个文件，打开对象关系设计工具。你会看到下面的界面：




对象关系设计工具正在显示People类，这是一个数据类，由SqlMetal.exe生成。它连接到数据库中的People表，并使用IEnumerable<T>接口返回数据，从而可以使用LINQ进行查询。真是太棒了！

**小提示：在非Express版本的Visual Studio中，可以跳过SqlMetal.exe，直接把SQL Server Compact Edition数据源拖入对象关系设计工具。**

**4 已经准备就绪，可以编写LINQ查询从数据库取出数据。**

将以下代码增加到`Main()`方法。注意这里使用了**`select new`**关键字来创建只包含Name和Company的定制结果。

```
string connectionString = "Data Source=|DataDirectory|\\ContactDB.sdf";
ContactDB context = new ContactDB(connectionString);

var peopleData =
    from person in context.People
    select new { person.Name, person.Company };

foreach (var person in peopleData)
    Console.WriteLine("{0} works at {1}", person.Name, person.Company);
```

ContactDB类称为数据上下文类。在LINQ查询中可以使用其People属性得到People表中的数据。

具体使用select new。它只从数据库取出Name和Company列的值。

## BULLET POINTS

- **group**子句告诉LINQ对结果分组，使用这个子句时，LINQ会创建组序列的一个序列。
- 每个组包含一个共同的成员，称为组键。使用**by**关键字来指定组键。每个组序列有一个**Key**成员，其中包含这个组的组键。
- 使用**join**子句可以告诉LINQ将两个集合合并到一个查询。此时，LINQ将把第一个集合中的各个成员与第二个集合中的各个成员比较，将匹配的各对放入结果集合。
- 连接查询使用一个**on...equals**子句来告诉LINQ如何匹配两个集合中的各对成员。
- 完成一个连接查询时，通常希望得到的结果中包含第一个集合中的一些成员，另外包含第二个集合中的另外一些成员。**select new**子句允许你从两个集合得到定制结果。
- LINQ可以使用**LINQ to SQL**类查询SQL数据库。这些类可以为使用LINQ的程序提供对象（这说明如果你愿意，可以直接访问这些对象的方法，你可以自己试试看）。
- 利用IDE的对象关系设计工具，可以选择希望通过LINQ访问哪些表。指定了要访问的表之后，它会向工程增加一个DataContext类。实例化这个类后，在LINQ查询中增加这个类实例的成员就可以访问SQL表。

## there are no Dumb Questions

**问：** 先等等，能不能再解释一下什么是var?

**答：** 当然可以。var关键字解决了LINQ带来的一个难题。正常情况下，调用一个方法或执行一个语句时，对于所处理的类型会很清楚。例如，如果有一个返回string的方法，就可以把它的结果保存在一个string变量或字段中。

但是LINQ没有这么简单。构建一个LINQ语句时，通常会返回程序中尚未定义的一个类型。没错，你知道这可能是某种序列。但是究竟是什么序列呢？你无从得知，因为这个序列中包含的对象完全取决于你在LINQ查询中选择了什么。例如，以Jimmy程序中的以下查询为例：

```
var mostExpensive =
  from comic in comics
  where values[comic.Issue] > 500
  orderby values[comic.Issue] descending
  select comic;
```

如果把最后一行改为：

```
select new
 { Name = comic.Name,
   IssueNumber = "#" + comic.Issue };
```

这会返回一个完全合法的类型。一个包含两个成员的匿名类型：分别是名为Name的string和名为IssueNumber的string。不过在我们的程序根本没有这个类型的类定义！确实，并不需要具体运行这处程序来看这个类型是如何定义的。不过mostExpensive变量还是需要声明为某种类型。

所以，C#提供了var关键字，它表示，“是的，我们知道这是一个合法的类型，但是现在还不能准确地告诉你它究竟是什么类型。所以你何不自己来确定，不要麻烦我们，多谢了。”

## there are no Dumb Questions

**问：** **我不太清楚究竟如何连接。**

**答：** 连接要处理两个序列。假设有一个足球球员集合players，它的各个成员分别是球员对象，包括一个Name属性、一个Position属性和一个Number属性。所以可以利用下面这个查询取出球衣号大于10的所有球员：

```
var results =
 from player in players
 where player.Number > 10
 select player;
```

下面假设我们希望确定每个球员的球衣尺寸，而且已经有一个jerseys集合，其中各项包含一个Number属性和一个Size属性。这里join就能起作用了：

```
var results =
 from player in players
 where player.Number > 10
 join shirt in jerseys
    on player.Number
    equals shirt.Number
 select shirt;
```

**问：** **等一下，这个查询会得到一组球衣。如果我想让每个球员与他的球衣大小连接，而不关心球衣上的号，该怎么做呢？**

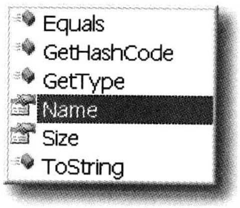
**答：** 这正是匿名类型的工作。可以构造一个匿名类型，其中只包含你想要的数据。而且还可以从所连接的不同集合选择。

这样就可以选择球员的名字和球衣大小，除此以外不包括其他内容：

```
var results =
 from player in players
 where player.Number > 10
 join shirt in jerseys
    on player.Number
    equals shirt.Number
 select new {
      player.Name,
      shirt.Size
 };
```

IDE很聪明，可以准确地知道你要用查询创建什么结果。如果创建一个循环来循环处理结果，一旦在IDE中键入变量名，它就会弹出一个智能提示窗口给出一个列表。

```
foreach (var r in results)
   r.
```




注意这个列表中包含有Name和Size。如果向select子句中增加更多项，它们也会显示在这个列表中。这是因为查询会创建一个包含不同成员的不同匿名类型。

**问：** **一定要增加SqlMetal.exe生成的那个.dbml文件吗？我还不太清楚到底发生了什么。**

**答：** 是的，如果你想对你的SQL Server Compact数据库使用LINQ，那么确实需要这个文件。

要记住，LINQ需要一个实现了IEnumerable<T>接口的对象。SQL数据库通常不会实现这个接口……实际上，它不会实现任何接口，因为数据库根本就不是一个对象。所以，如果你想对SQL数据库（或任何其他可以查询的数据源）使用LINQ，就需要一个与LINQ交互的对象，而且要实现IEnumerable<T>。

不过不要光听我们的一面之辞。进入你刚编写的代码，右键点击“People”，并选择“Go to Definition”（或者按下F12）。这会带你进入一个ContactDB.designer.cs获取存取方法，它会返回一个Table<People>。点击“Table”，再进入它的定义。类Table<TEntity>扩展了IQueryable<TEntity>。点击“IQueryable”，再次进入定义，你会看到它确实实现了IEnumerable<T>。

所以这个.dbml文件（和一同生成的.cs类文件）提供了实现IEnumerable的对象。IDE非常聪明，它很清楚地知道该对这个.dbml文件做什么：生成这个文件时，把它增加到你的工程，在对象关系设计工具中打开这个文件，你会看到People数据类的成员直接映射到SQL数据库中的People表。这个数据类会为你完成连接，它会自动读入数据库的表，并把数据包装在一个IEnumerable<T>中以便LINQ访问。

**可以使用“select new”构造定制LINQ查询结果，其中只包括你希望放在结果序列中的项。**

# 使用一个join查询连接Starbuzz和Objectville

现在已经有了所需的全部工具，可以将Starbuzz和Objectville纸业公司的数据合并到最终的一个结果集中。

**动手做**

1. **向工程增加SQL数据。**
完成后，创建一个新的控制台应用工程，并为工程增加一个ContactDB SQL数据库。然后使用SqlMetal.exe创建对象关系设计工具，为工程提供LINQ to SQL类，把它增加到工程，并编写一个简单的测试查询来确保它能正确工作。

2. **构建Starbuzz对象。**
下面的列表中包含Starbuzz顾客数据。把它们增加到你的工程中：

```
class StarbuzzData {
    public string Name { get; set; }
    public Drink FavoriteDrink { get; set; }
    public int MoneySpent { get; set; }
    public int Visits { get; set; }
}
enum Drink {
    BoringCoffee, ChocoRockoLatte, TripleEspresso,
    ZestyLemonChai, DoubleCappuccino, HalfCafAmericano,
    ChocoMacchiato, BananaSplitInACup,
}
```

Starbuzz数据作为一个StarbuzzData对象集合存储。其中包含大量数据，对于这个推广活动来说并不需要所有这些数据，因此必须在LINQ查询中只选择所需的数据。

Starbuzz有很多很棒的饮品，每个顾客都有自己的最爱。

还需要一个方法生成一些示例数据：

```
static IEnumerable<StarbuzzData> GetStarbuzzData() {
    return new List<StarbuzzData> {
        new StarbuzzData {
            Name = "Janet Venutian", FavoriteDrink = Drink.ChocoMacchiato,
            MoneySpent = 255, Visits = 50 },
        new StarbuzzData {
            Name = "Liz Nelson", FavoriteDrink = Drink.DoubleCappuccino,
            MoneySpent = 150, Visits = 35 },
        new StarbuzzData {
            Name = "Matt Franks", FavoriteDrink = Drink.ZestyLemonChai,
            MoneySpent = 75, Visits = 15 },
        new StarbuzzData {
            Name = "Joe Ng", FavoriteDrink = Drink.BananaSplitInACup,
            MoneySpent = 60, Visits = 10 },
        new StarbuzzData {
            Name = "Sarah Kalter", FavoriteDrink = Drink.BoringCoffee,
            MoneySpent = 110, Visits = 15 }
    };
}
```

GetStarbuzzData()使用一个集合初始化方法和对象初始化方法来创建Starbuzz对象。

再次说明，可以从集合和对象初始化方法去掉()。

建立这个方法，这样就能有一些人名同时也出现在Objectville联系表中。如果你使用的名字不同，一定要确保这里有匹配的数据。

**3 现在把SQL数据库连接到Starbuzz集合。**

以下是这个查询的代码，把它放在你的Main()入口点方法中：

```
IEnumerable<StarbuzzData> starbuzzList = GetStarbuzzData();

string connectionString =
     "Data Source=|DataDirectory|\\ContactDB.sdf";
ContactDB context = new ContactDB(connectionString);

var results =
   from starbuzzCustomer in starbuzzList
   where starbuzzCustomer.MoneySpent > 90
   join person in context.People
   on starbuzzCustomer.Name equals person.Name
   select new { person.Name, person.Company,
                starbuzzCustomer.FavoriteDrink };

foreach (var row in results){
    Console.WriteLine("{0} at {1} likes {2}",
                row.Name, row.Company, row.FavoriteDrink);

Console.ReadKey();
```

需要完成一个连接来合并Starbuzz数据与People表中的客户数据。

在这里select子句从数据库取出人名和所在公司，并从Starbuzz数据取出顾客最喜欢的饮品，把它们放在一个结果序列中。

DataContext中的People成员是一个集合，利用这个集合可以访问数据库中的People表。

检查你的结果，确保一切正常。

## 用LINQPad编辑查询

有一个很棒的学习工具可以帮助你研究和使用LINQ。这就是LINQPad，可以从Joe Albahari免费获得（他是我们的一位技术审校，为确保“Head First C#”尽可能没有错误做出了很多贡献）。可以从这里下载这个工具：

**http://www.linqpad.net/**



姓名：　　　　日期：

# C#实验室

# 入侵者

这个实验室将给出一个程序的规范，要求你根据前几章所学的知识构建这个程序。

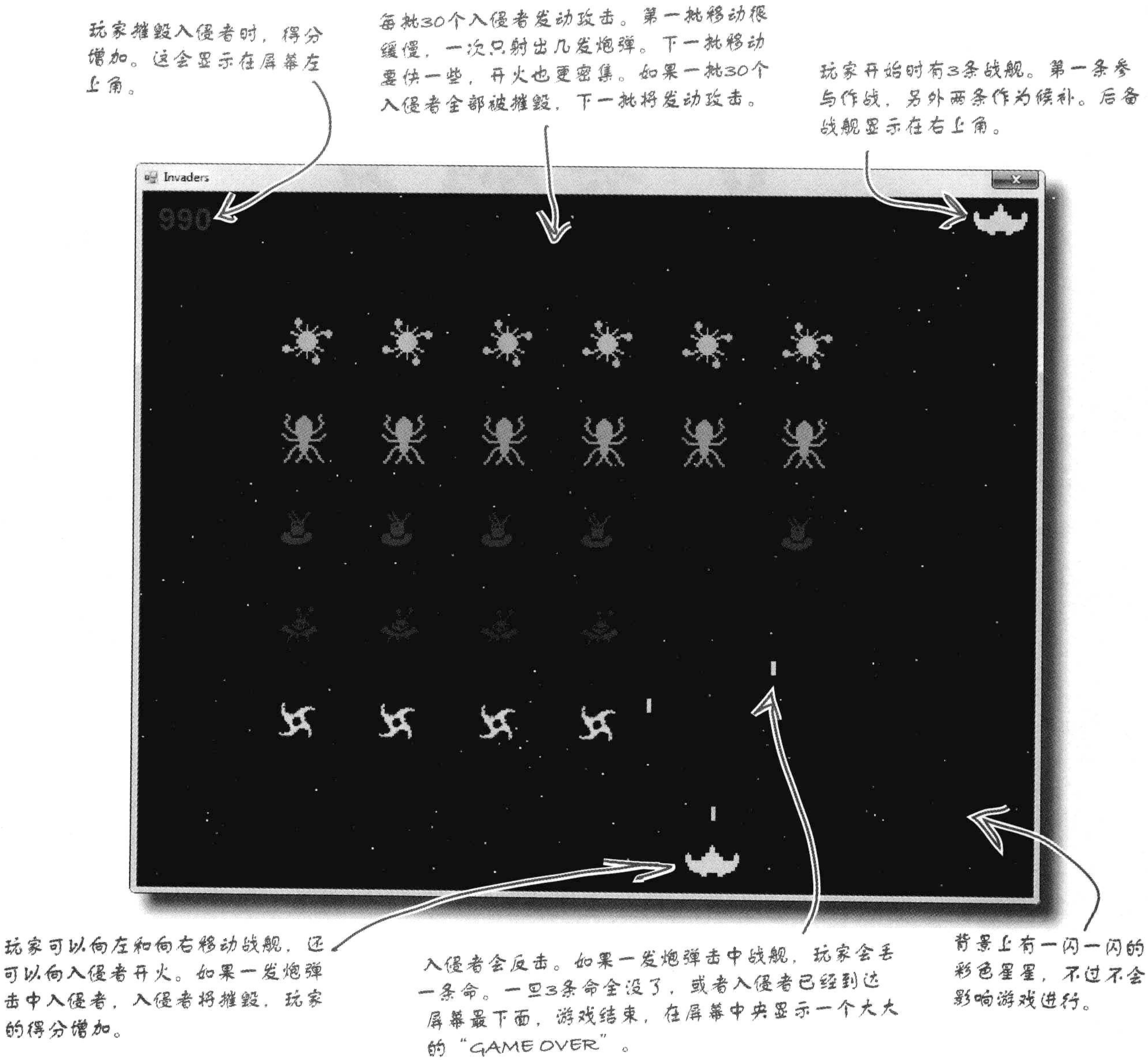
这个工程比你之前见过的所有工程规模更大。所以在着手开发之前，先花点时间了解整个问题，磨刀不误砍柴工。另外，如果你被问题卡住了也不用担心，这里并没有新的内容，所以完全可以继续向下读，以后再回来完成这个实验室。

我们已经为你完成了一些设计细节，而且可以保证已经提供了你需要的全部知识……再没有其他可以补充的了。

**你要自己来完成这个任务。**可以从网站下载这个实验室的一个可执行文件…… 不过我们不会提供这个工程的源代码。

# 经典的视频游戏

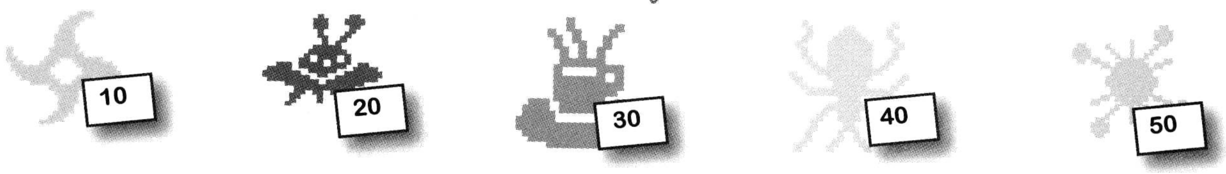
在这个实验室里，你将回顾视频游戏历史上曾经最游行、最受人推祟的游戏之一，这个游戏不需要多做介绍了。下面来构建这个鼎鼎大名的入侵者游戏——Invaders。



## 你的任务：保卫你的星球，防范成批进攻的入侵者

入侵者成批发动攻击，每一批30个入侵者构成一个紧密编队。玩家摧毁入侵者时，得分会增加。最下面的入侵者形状类似星星，值10分。飞船值20分，茶杯值30分，小虫值40分，卫星值50分。玩家开始时有3条命。如果三条命都丢了，或者入侵者到达屏幕最下方，游戏结束。

有5种不同类型的入侵者，不过它们的行为都一样。先从屏幕最上方出发，向左移，直到到达边界，然后向下落，并开始向右移。到达右边界时，又向下落，再次向左移。如果入侵者到达屏幕最下方，游戏就结束了。



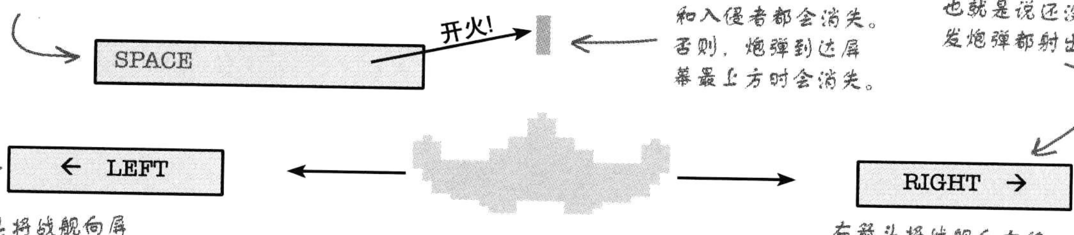

第一批入侵者一次能弹出两发炮弹，如果屏幕上已经有两发炮弹，入侵者会停止开火。下一批一次能射出3发炮弹，再下一批可以射出4发炮弹，依此类推。



空格是开火键。不过屏幕上一次只能有两发炮弹。一旦一发炮弹击中某个目标或消失，就可以发射另一发炮弹。

如果一发炮弹击中一个入侵者，炮弹和入侵者都会消失。否则，炮弹到达屏幕最上方时会消失。

游戏会跟踪所有按键情况。所以按下右箭头和空格会让战舰向右移并开火（如果屏幕上还没有两发炮弹，也就是说还没有将两发炮弹都射出）。



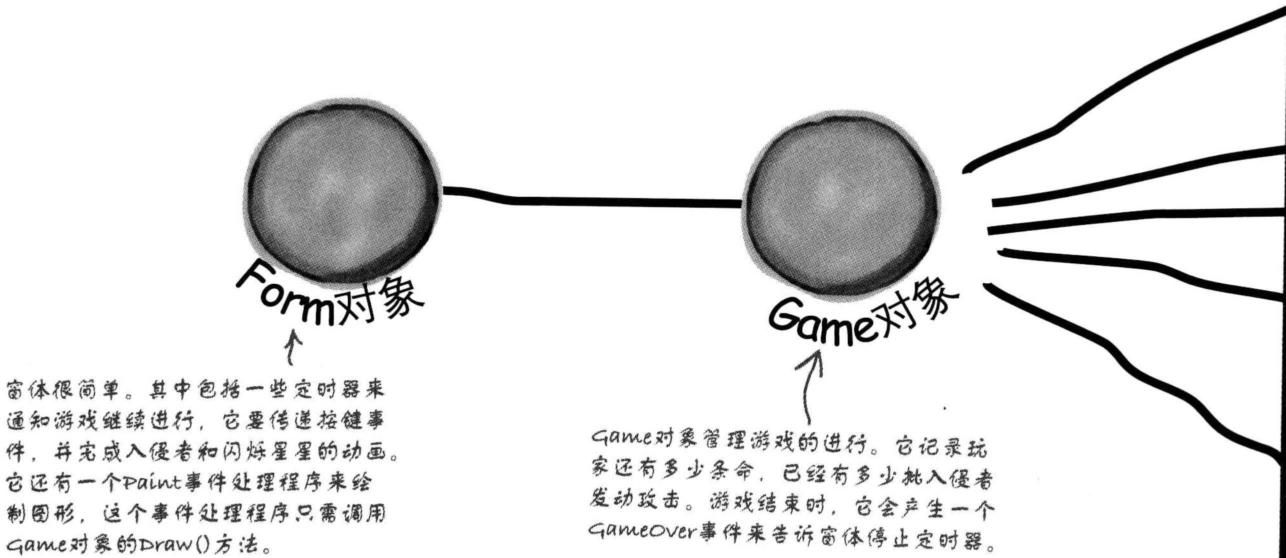

左箭头将战舰向屏幕左边界移动。

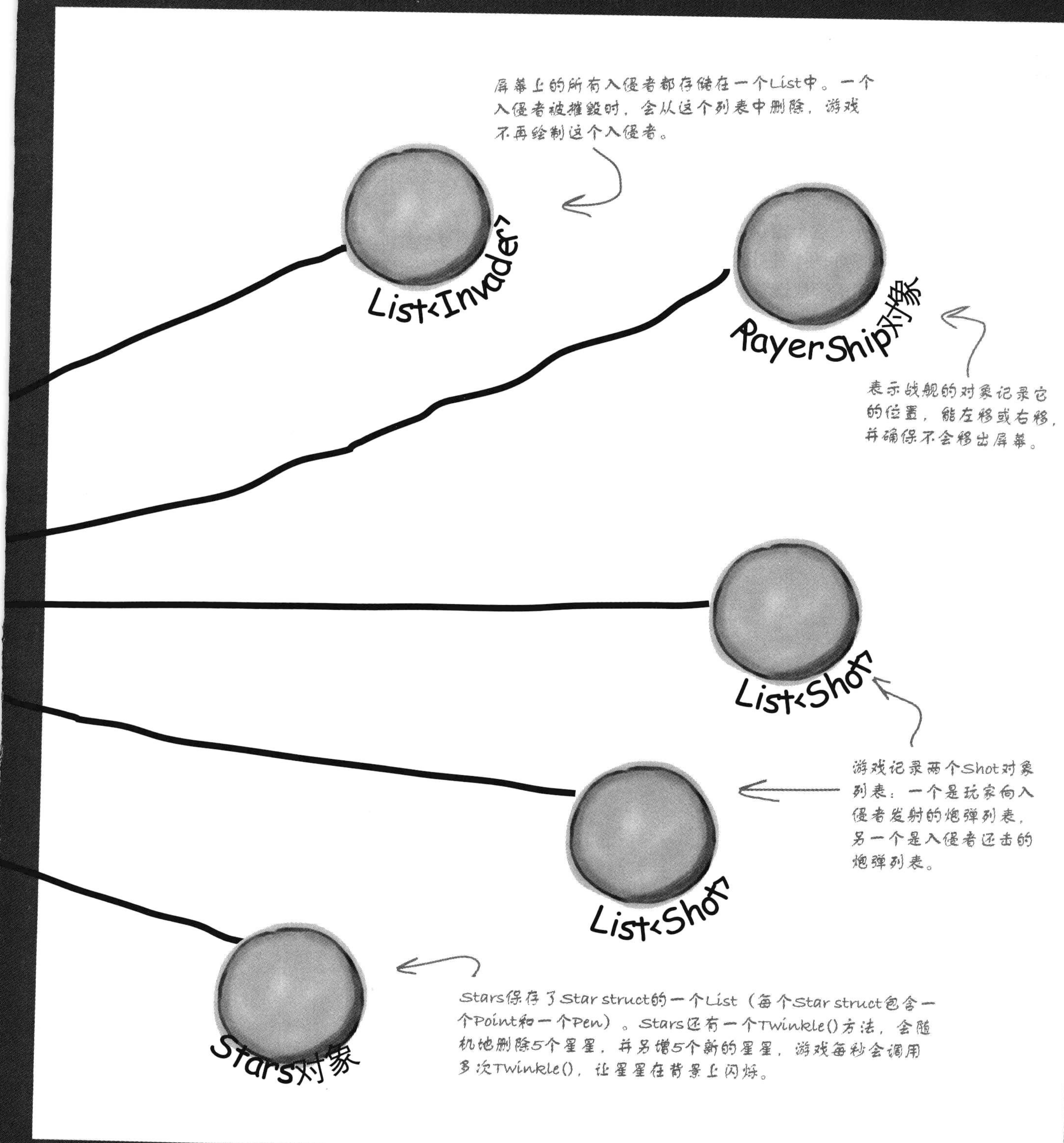
右箭头将战舰向右移。

# 入侵者体系结构

入侵者游戏需要记录一批30个入侵者（包括它们的位置、类型和分值）、玩家的战舰，玩家和入侵者相互发射的炮弹，以及背景上的星星。与冒险游戏实验室中一样，需要一个Game对象来记录所有这些内容，并完成窗体与这些游戏对象之间的协调。

以下对要创建的对象做一个概要说明：



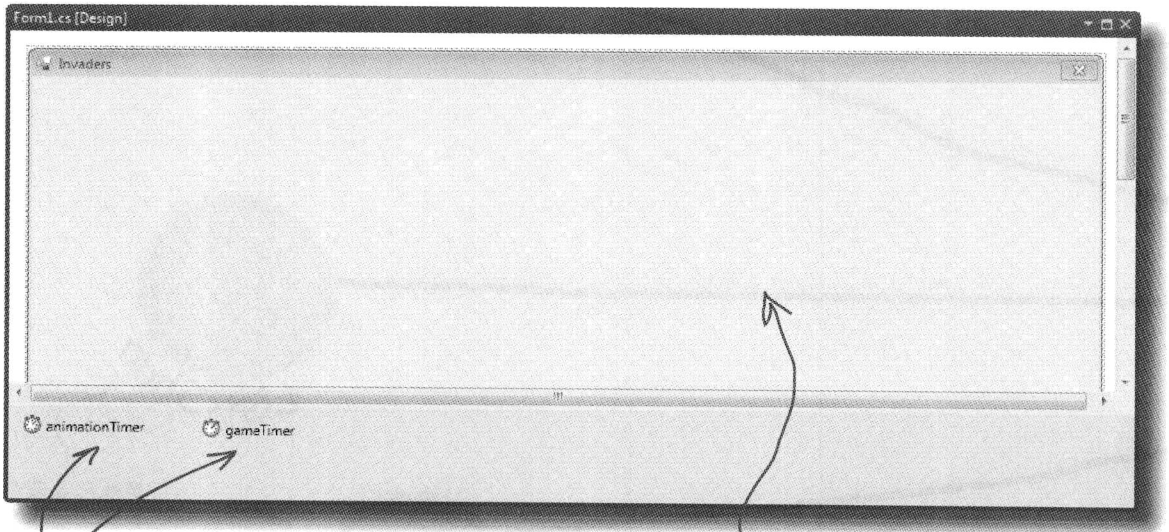

屏幕上的所有入侵者都存储在一个List中。一个入侵者被摧毁时，会从这个列表中删除，游戏不再绘制这个入侵者。
List<Invader>
PlayerShip对象
表示战舰的对象记录它的位置，能左移或右移，并确保不会移出屏幕。
List<Shot>
游戏记录两个Shot对象列表：一个是玩家向入侵者发射的炮弹列表，另一个是入侵者还击的炮弹列表。
List<Shot>
Stars对象
Stars保存了Star struct的一个List（每个Star struct包含一个Point和一个Pen）。Stars还有一个Twinkle()方法，会随机地删除5个星星，并另增5个新的星星，游戏每秒会调用多次Twinkle()，让星星在背景上闪烁。

## 设计入侵者窗体

入侵者窗体只有两个控件：一个是触发动画的定时器（保证星星闪烁，并实现入侵者动画，这是通过将各个入侵者图片切换为另一帧实现的），另一个定时器控件用于控制游戏的进行（入侵者左右行进，玩家移动，玩家和入侵者相互开火）。除此以外，这个窗体中惟一有点难度的是一个处理游戏GameOver事件的事件处理程序，以及KeyUp和KeyDown事件处理程序来管理键盘输入。

只要按下键，窗体就会触发一个KeyDown事件，松开按键时则触发一个KeyUp事件。

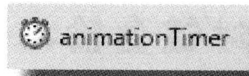
窗体初始化Game对象时，会传入窗体的ClientRectangle，使Game对象知道窗体的边界。因此，可以通过改变窗体的大小来改变战场的大小。




要增加两个定时器：animationTimer和gameTimer。

设置窗体的FormBorderStyle属性为FixedSingle，DoubleBuffered属性设置为true，将其MinimizeBox和MaximizeBox属性置为false，设置其标题，然后拉伸窗体，达到你所希望的游戏战场宽度。

# 动画定时器处理视觉效果




游戏背景上的星星和入侵者动画不会影响游戏进行，游戏暂停或停止时这些动画仍会继续。所以需要为它们建立一个单独的定时器。

## 为动画定时器的tick事件增加代码

这里应当有一个计数器，从0到3，然后循环回到0。这个计数器用于更新各个入侵者动画（各动画包含4帧或4个单元），来创建一个平滑的动画。事件处理程序还应当调用Game对象的Twinkle()方法，这会使星星闪烁。最后，需要调用窗体的Refresh()方法重绘屏幕。

即使游戏没有运行，这些动画仍会继续。这说明即使游戏结束、暂停或者根本没有开始，星星也会闪烁，入侵者动画仍会进行。

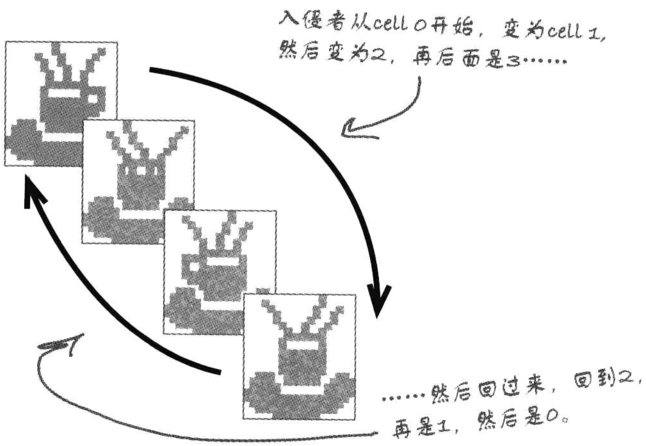
将这个定时器间隔设置为33ms，这样每秒约有30帧。不过，游戏定时器一定要设置为一个更短的间隔。战舰移动和游戏运行的速度应该比星星闪烁快一些。

## 调整定时器完成平滑的动画

动画间隔时间为33ms，将游戏定时器间隔设置为10ms。这样一来，游戏的速度就会比动画快（这些动画实际上只是背景视觉效果）。另外，Game中的Go()方法（由游戏定时器触发，稍后就会讨论游戏定时器）可能会占用大量CPU周期。如果CPU忙于处理游戏，动画定时器只能等待，直到CPU有机会处理动画，然后触发动画定时器（完成星星和入侵者的动画）。

如果动画定时器设置为33ms，但是运行Game对象的Go()方法需要更长时间，那么一旦Go()调用结束就会运行动画。

或者，也可以将两个定时器都设置为间隔为5ms，游戏就能按系统所能处理的最快速度运行以及完成动画（不过，如果机器速度很快，动画可能会过快而让人反感）。




我们在一个速度很慢的机器上做了一个尝试，将动画间隔设置为100ms，游戏定时器间隔设置为50ms，此时帧速率大约为每秒10帧，我们发现这是完全可以接受的。可以以此作为起点，适当地缩小这两个定时器的间隔，直到满意为止。

# 对键盘输入的响应

编写游戏定时器的代码之前，需要为KeyDown和KeyUp事件编写事件处理程序。按下键时会触发KeyDown，放开键时将触发KeyUp。大多数按键可以直接采取行动，如开火或退出游戏。

对于某些按键（如左右箭头），我们希望将这些按键保存到一个列表中，以便游戏定时器用来移动玩家的战舰。所以窗体对象中还需要一个保存按键的列表：

所以如果玩家同时按下左箭头和空格，列表中将包含Keys.Left和Keys.Space。

需要一个按键列表，从而跟踪按下了哪些键。稍后，游戏定时器需要这个列表来完成移动。

```
List<Keys> keysPressed = new List<Keys>();

private void Form1_KeyDown(object sender, KeyEventArgs e) {
  if (e.KeyCode == Keys.Q)
    Application.Exit();

  if (gameOver)
    if (e.KeyCode == Keys.S) {
      // code to reset the game and restart the timers
      return;
    }

  if (e.KeyCode == Keys.Space)
    game.FireShot();
  if (keysPressed.Contains(e.KeyCode))
    keysPressed.Remove(e.KeyCode);
  keysPressed.Add(e.KeyCode);
}

private void Form1_KeyUp(object sender, KeyEventArgs e) {
  if (keysPressed.Contains(e.KeyCode))
    keysPressed.Remove(e.KeyCode);
}
```

'Q'按键会退出游戏。

Keys enum定义要根据哪些按键检查按键码。

如果游戏已经结束，重置游戏，并再次启动。

不过希望只在游戏已经结束时这个按键才起作用。如果程序正在运行，按下S不会让游戏重启。

需要由你来填写这部分代码。

空格发射一发炮弹。

删除这个按键，然后重新增加，使得这个按键成为列表中最后（最新）的一项。

按键会增加到key列表，稍后就会用到。

我们希望最后按下的按键位于列表的最前面，这样一来，如果玩家同时按下多个键，游戏会对最后的按键做出响应。放开一个按键时，游戏会对列表中的下一个按键做出响应。

放开一个按键时，把它从按键列表中删除。

**翻回到第4章的KeyGame工程。那里也使用了一个KeyDown事件处理程序！**

## 游戏定时器处理移动和游戏运行

gameTimer

窗体游戏定时器的主要任务是调用Game类中的Go()。不过，它还会对按键做出响应，因此必须检查keysPressed列表，查找KeyDown和KeyUp事件捕获的按键：

这个定时器使游戏前进一帧。所以它首先会调用Game对象的Go()方法让游戏继续。

玩家会同时“压下”很多个按键。如果我们希望游戏尽可能健壮，它就应当能处理这种情况。正是因为这个原因我们使用了keysPressed列表。

```
private void gameTimer_Tick(object sender, EventArgs e)
{
    game.Go();
    foreach (Keys key in keysPressed)
    {
        if (key == Keys.Left)
        {
            game.MovePlayer(Direction.Left);
            return;
        }
        else if (key == Keys.Right)
        {
            game.MovePlayer(Direction.Right);
            return;
        }
    }
}
```

keysPressed是KeyDown和KeyUp事件处理程序管理的List<Keys>对象。其中包含玩家目前按下的每一个按键。

keysPressed列表根据按键的顺序包含所有按键。这个foreach循环会循环处理各个按键，直到找到左键（Left）或右键（Right），此时会移动玩家并返回。

KeyUp和KeyDown事件使用Keys enum来指定一个按键。我们将使用Keys.Left和Keys.Right来移动战舰。

```
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
```

炮弹上移或下移，玩家左移和右移，入侵者左移、右移和下移。都需要这个enum来确定方向。

KeyDown事件处理程序只处理空格、S和Q键，而没有把它们增加到keysPressed列表。如果把按下空格时触发炮弹发射的代码移到这个事件处理程序中会怎么样呢？

## 窗体的另一个细节问题：GameOver事件

向窗体增加一个私有bool字段，名为gameOver，只有当游戏结束时这个字段才为true。然后为Game对象的GameOver事件增加一个事件处理程序，停止游戏定时器（但是不停止动画定时器，这样星星还会闪烁，并且继续显示入侵者动画），设置gameOver为true，并调用窗体的Refresh()方法.

这里向窗体增加另一个事件但是并不使用IDE来增加。这个事件处理程序的代码都由手工编写。

游戏结束事件及其委托放在Game类中，稍后就会看到。

编写窗体的Paint事件处理程序时，要检查gameOver。如果它为true，必须在屏幕中央写上大大的黄色字母“GAME OVER”。然后在右下角写出“Press S to start a new game or Q to quit”。在这个状态下可以启动游戏，所以用户必须按S来开始一个新游戏。

# 窗体的游戏定时器告诉游戏调用Go()

除了处理左移和右移，游戏定时器的主要任务是调用Game对象的Go()方法。游戏的所有工作都在这里处理。Game对象要记录游戏的状态，它的Go()方法逐帧地推进游戏进行。这包括：

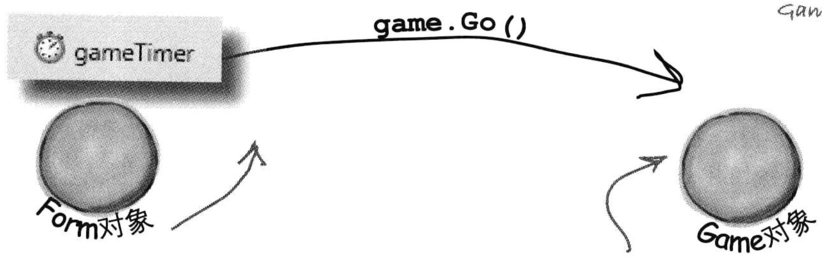
1. 使用玩家的Alive属性查看玩家是不是还活着。如果玩家已死，游戏会显示一个小动画，表现战舰被击毁的过程（使用DrawImage()将战舰击毁）。这个动画由PlayerShip类完成，所以Go()只需要查看它是否已死。如果确实已死则直接返回，这样一来，入侵者就不会继续移动或开火，使玩家有一个短暂的间歇可以看到他的战舰被击毁的情形）。

2. 移动各发炮弹。入侵者射出的炮弹向下移，玩家射出的炮弹向上移。Game保存了两个List<Shot>对象，一个存储入侵者的炮弹，另一个保存玩家的炮弹。移出屏幕的炮弹要从列表中删除。

3. 移动各个入侵者。Game调用各个Invader对象的Move()方法，告诉入侵者向哪个方向移动。Game还会记录入侵者在什么位置，以便向下移一行或者换方向。之后，Game查看是否轮到入侵者还击，如果是，则向List<>增加新的Shot对象。

4. 检查是否击中。如果玩家的一发炮弹击中某个入侵者，Game从相应的List<>删除这个入侵者。然后Game查看是否有某个入侵者的炮弹打中玩家的战舰，如果是，则杀死玩家，将其Alive属性设置为false。如果玩家3条命都没了，Game会产生GameOver事件告诉窗体游戏结束。窗体的GameOver事件处理程序停止其游戏定时器，所以不再调用Go()。

这里要用到上一页编写的GameOver事件处理程序。



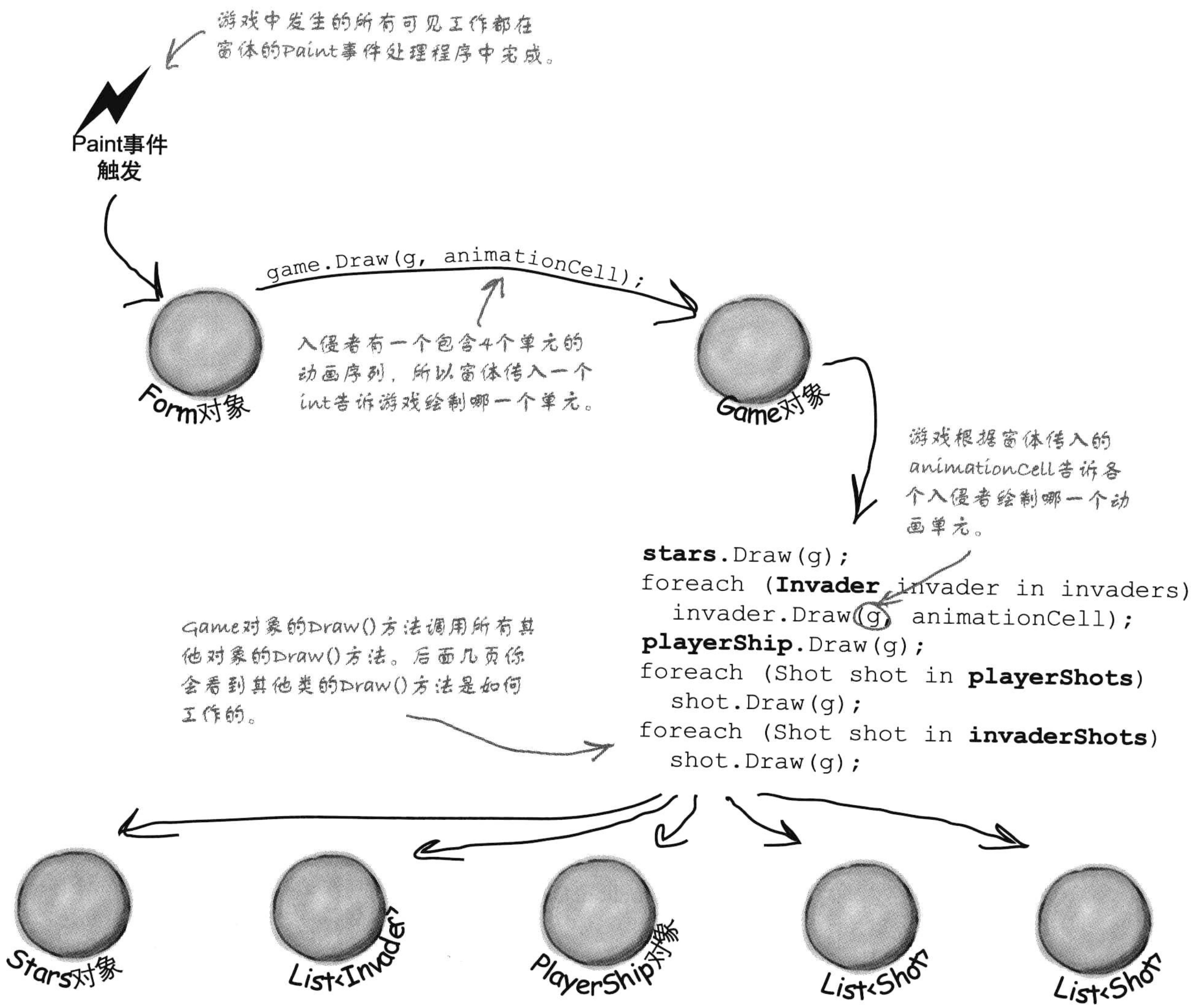

游戏定时器比动画定时器触发的频率更快，使游戏快速进行。

一切都由Game对象中的Go()处理，从移动到开火，以及查看战舰或入侵者是否被击中。

## 控制图形化

在前面的实验室里，窗体都使用控件来完成图形化。但是现在你已经知道了如何使用Graphics和双缓冲，应该让Game对象来处理大量绘图工作。

所以窗体应当有一个Paint事件处理程序（一定要把窗体的`DoubleBuffered`属性设置为true）。每次触发窗体的Paint事件时，通过调用`Game`的`Draw()`方法，其余绘图工作将委托Game对象完成。



# 构建Game类

Game类是入侵者游戏的控制器。可以把以下代码作为这个类的起点，不过还有很多工作需要你来完成。

```
class Game {
  private int score = 0;
  private int livesLeft = 2;
  private int wave = 0;
  private int framesSkipped = 0;

  private Rectangle boundaries;
  private Random random;

  private Direction invaderDirection;
  private List<Invader> invaders;

  private PlayerShip playerShip;
  private List<Shot> playerShots;
  private List<Shot> invaderShots;

  private Stars stars;


  public event EventHandler GameOver;

  // etc...
}
```

score, livesLeft和wave字段记录有关游戏状态的一些基本信息。

将使用framesSkipped字段减慢游戏中入侵者的行动，第1批入侵者左移之前要跳过6帧，下一批应当跳过5帧，再下一批应当跳过4帧，依此类推。

这个Invader对象List<>记录当前这一批中的所有入侵者。一个入侵者被摧毁时，会从列表中将其删除。游戏会定期进行检查，确保列表不为空，如果列表为空，则出现下一批入侵者。

Stars对象跟踪背景上的彩色星星。

玩家已死，而且3条命都没有了，此时Game对象产生GameOver事件。要在窗体中建立事件处理方法，并关联到Game对象的GameOver事件。

| Game |
|---|
| GameOver: event |
| Draw(g: Graphics, animationCell: int)<br>Twinkle()<br>MovePlayer(direction: Direction)<br>FireShot()<br>Go() |

其中大多数方法都要结合其他对象的方法，来完成一个特定的动作。

要记住，这些是公共方法。你的代码中可能还需要大量私有方法，合理地建立代码结构。

# Game类方法

Game类有5个公共方法，由窗体中发生的不同事件触发。

1. **Draw()方法在一个Graphics对象上绘制游戏。**
   Draw()方法有两个参数：一个Graphics对象和一个整数，这个整数对应一个动画单元（这是一个0到3之间的数）。首先，它要画一个黑色矩形占满整个窗体（使用boundaries中存储的显示矩形，这个boundaries由窗体得到）。然后这个方法要画出星星、入侵者，然后是玩家的战舰，再画出炮弹。最后，它要在左上角画出得分，并在右上角画出玩家的候补战舰，如果gameOver为true，还在用黄色字母画出一个大大的“GAME OVER”。

2. **Twinkle()方法让星星闪烁。**
   窗体的动画定时器事件处理程序要让星星闪烁，所以Game对象需要一个只有一行代码的方法来调用`stars.Twinkle()`。 ← 后面几页会编写Stars对象的代码。

3. **MovePlayer()方法让玩家移动。**
   窗体的键盘定时器事件处理程序需要移动玩家的战舰，所以`Game`对象还需要一个包含两行代码的方法，取一个`Direction enum`为参数，先检查玩家是不是已经死了，并调用`playerShip.Move()`影响这一次移动。

4. **FireShot()方法向入侵者开火。**
   `FireShot()`方法查看屏幕上玩家的炮弹是否少于两发。如果是，这个方法会在适当的位置向`playerShots`列表增加一发新炮弹。

5. **Go()方法使游戏继续进行。**
   窗体的游戏定时器。以一定频率调用Game对象的Go()方法，每秒10到30次（取决于计算机的CPU速度）。Go()方法完成游戏向前推进一帧要做的所有工作：

   - ★ 游戏使用玩家的Alive属性检查它是否已经死了。如果还活着，游戏就不结束，如果已死，窗体会用其`Stop()`方法停止游戏定时器。所以除非玩家还有命，否则`Go()`方法不会做任何事情，只是简单地返回。
   - ★ 每个炮弹都需要更新。游戏需要循环处理两个`List<Shot>`对象，调用各个炮弹的`Move()`方法。如果某个炮弹的`Move()`返回`false`，这说明炮弹飞出了屏幕边界，所以会从列表中删除。
   - ★ 游戏移动各个入侵者，并允许它们还击。
   - ★ 最后检查击中情况：首先检查是否有炮弹与某个入侵者重叠（将炮弹和入侵者从各自的`List<T>`对象删除），然后查看玩家是否被打中。我们将为`Invader`和`PlayerShip`类增加一个`Rectangle`属性，名为`Area`，这样就可以使用`Contains()`方法来检查战舰区域是否与炮弹重叠。

# 填写Game类

类图存在一个问题，它们往往没有显示出非公共属性和方法。所以即使已经有了725页所列的方法，还有很多工作要做。以下是需要考虑的一些问题：

## 构造函数完成初始创建

Game对象需要创建所有其他对象，包括Invader对象、PlayerShip对象、保存炮弹的List对象以及Stars对象。窗体传入一个初始化Random对象和它自己的DisplayRectangle struct（以便Game确定战场的边界，它要用这个矩形来确定炮弹是否飞出范围，以及入侵者何时到达边界需要下移并改变方向）。你的代码需要创建这个游戏世界中的所有其他对象。

在这个实验室的后面几页我们将讨论以上各个对象。

## 构建一个NextWave()方法

如果有一个创建下一批入侵者的简单方法，这会很有用。它要向invaders字段指定一个新的`Invader`对象List，并增加6列共30个入侵者，使它们位于其起始位置，将wave字段增1，并设置`invaderDirection`字段，使它们朝着屏幕右侧移动。还要改变`framesSkipped`字段。

私有方法对于Game类的组织确实很有帮助，这就是一个例子。

## 关于私有方法的另外一些想法

以下是另外一些关于私有方法的想法，供你参考，可以看看这些是对于你设计Game类是否有帮助：

- ✓ 查看玩家是否被击中的方法(`CheckForPlayerCollisions()`)
- ✓ 查看入侵者是否被击中的方法 (`CheckForInvaderCollisions()`)
- ✓ 移动所有入侵者的方法 (`MoveInvaders()`)
- ✓ 允许入侵者还击的方法(`ReturnFire()`)

**BRAIN POWER**

在类图上也可以显示保护和私有属性及方法，但是在实际中很少看到这样做，你认为这是为什么？

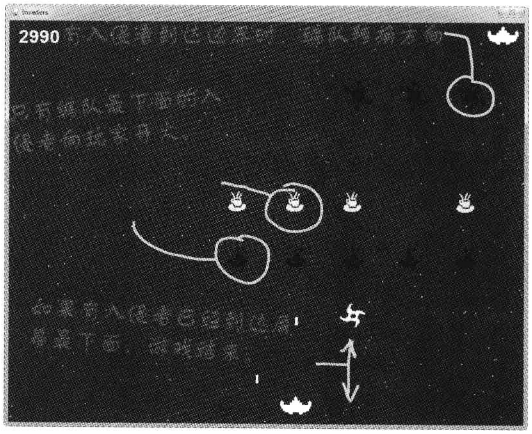
# 利用LINQ使碰撞检测更容易

你已经有了入侵者和炮弹的集合，而且需要搜索这些集合来查找某些入侵者和炮弹。只要同时听到集合和搜索，就应该想到LINQ。你需要完成以下工作：

刚开始可能会认为这相当复杂，但是每个LINQ查询实际上只有几行代码。给你一个提示：不要过分复杂化！

1. **确定入侵者编队是否到达战场的边界。**
   如果某个入侵者在距战场边界100个像素的范围内，入侵者就需要改变方向。入侵者向右行进时，一旦到达窗体的右边界，游戏就要告诉它们下移，并向左边行进。当入侵者向左行进时，游戏需要检查它们是否到达左边界。为此，需要增加一个私有`MoveInvaders()`方法，由`Go()`调用。首先它要检查并更新私有framesSkipped字段，如果应当跳过这一帧（取决于目前在第几关），则返回。然后检查入侵者朝着哪个方向移动。如果入侵者向右移，`MoveInvaders()`应当使用LINQ搜索`invaderCollection`列表中是否有某个入侵者位置的X值在右边界100像素范围内。如果找到这样的入侵者，则应当告诉入侵者下移，然后设置`invaderDirection`等于`Direction.Left`；如果没有找到，则告诉各个入侵者继续右移。另一方面，如果入侵者向左移，就应当做相反的工作，使用另一个LINQ查询查看是否有入侵者在左边界的100像素范围以内，如果有这样的入侵者，则要求它们向下移，并改变方向。

2. **确定哪些入侵者可以还击。**
   增加一个私有方法，名为ReturnFire()，由Go()调用。首先，如果入侵者的炮弹列表已经有 wave + 1发炮弹，则应当返回。如果random.Next(10) < 10 - wave也应当返回（这会使入侵者随机开火，而且不是每一次都开火）。如果通过了这两个测试，可以使用LINQ将入侵者按其Location.X分组，并按降序排序。一旦分组，可以随机选择一组，并使用其First()方法找出这一列最下面的入侵者。好了，现在就找到射手了，可以向入侵者炮弹列表中增加一发炮弹，显示在该入侵者下方的正中间（使用入侵者的Area来设置炮弹的位置）。




3. **检查入侵者和玩家的碰撞情况。**
   你可能希望创建一个方法来检查碰撞情况。需要检查3种碰撞，在这里Rectangle struct的Contains()方法非常有用，只需传入Point，如果这个点在该矩形范围内，就会返回true。

   ★ 使用LINQ循环处理玩家炮弹列表中的各发炮弹，并选择满足条件（invader.Area包含这发炮弹的位置）的入侵者，找出被打死的入侵者。将这个入侵者和这发炮弹删除。

   ★ 增加一个查询，确定入侵者是否到达屏幕最下面，如果是，则结束游戏。

   ★ 不需要LINQ来查找打中玩家的炮弹，只需要一个循环并使用玩家的Area属性（要记住，foreach循环中不能修改集合。如果尝试这样做，会得到一个`InvalidOperationException`异常，并有一个消息指出集合被修改）。

# 创建Invader类

Invader类记录一个入侵者。所以当Game对象创建一批新的入侵者时，它会向一个List<Invader>对象增加30个Invader实例。每次调用Game对象的Go()方法时，它会调用各个入侵者的Move()方法，通知各个入侵者移动。每次调用Game对象的Draw()方法时，则会调用各个入侵者对象的Draw()方法。所以需要为Invader类创建Move()和Draw()方法。另外，可能还需要增加一个私有方法，名为InvaderImage()，在绘制入侵者时这个方法会很有用。要在Draw()方法中调用这个InvaderImage()方法，保证image字段保持更新：

| Invader |
|---|
| Location: Point<br>InvaderType: ShipType<br>Area: Rectangle<br>Score: int |
| Draw(g: Graphics, animationCell: int)<br>Move(direction: Direction) |

```
class Invader {
  private const int HorizontalInterval = 10;
  private const int VerticalInterval = 40;

  private Bitmap image;

  public Point Location { get; private set; }

  public ShipType InvaderType { get; private set; }

  public Rectangle Area { get {
    return new Rectangle(location, image.Size); }
  }

  public int Score { get; private set; }

  public Invader(ShipType invaderType, Point location, int score) {
    this.InvaderType = invaderType;
    this.Location = location;
    this.Score = score;
    image = InvaderImage(0);
  }

  // Additional methods will go here
}
```

HorizontalInterval常量确定每次入侵者向左或向右行进时移动多少个像素。VerticalInterval是编队到达战场边界时下移的像素个数。

用Area属性来检查效果。由于我们知道入侵者的位置，而且知道它的大小（根据其image字段），所以可以增加一个获取存取方法计算所覆盖区域的相应Rectangle。

……这说明可以使用该Rectangle的Contains()方法利用一个LINQ查询检查是否有炮弹打中一个入侵者。

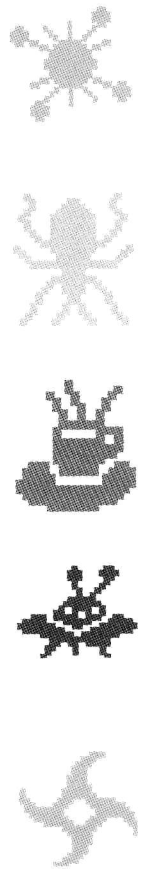
入侵者使用ShipType enum来确定这是哪一种类型的敌人。

```
enum ShipType {
    Bug,
    Saucer,
    Satellite,
    Spaceship,
    Star,
  }
```

# 创建Invader方法

Invader的3个核心方法是Move()、Draw()和InvaderImage()。下面来依次分析这几个方法：

有以下5种类型的入侵者，每一类入侵者都有4个不同的动画单元图片。



## 移动入侵者战舰

首先，需要一个移动入侵者战舰的方法。Game对象使用Direction enum指定一个方向，然后入侵者战舰移动。要记住，Game对象会确定入侵者是否需要下移或改变方向，所以Invader类不需要操心这个问题。

```
public void Move(Direction direction) {
  // This method needs to move the ship in the
  //   specified direction
}
```

## 绘制战舰和适当的动画单元

每个Invader知道如何自行绘制。给定一个Graphics对象（将在这个对象上绘制），并给定要使用哪一个动画单元，入侵者可以使用Game提供的Graphics对象在游戏面板上完成绘制。

```
public void Draw(Graphics g, int animationCell) {
  // This method needs to draw the image of
  //   the ship, using the correct animation cell
}
```

## 得到适当的Invader图片

由于经常需要根据动画单元得到适当的图像，所以可能希望把这部分代码放在一个单独的方法中。为此建立一个InvaderImage()方法，给定一个动画单元时它会返回一个特定的Bitmap。

```
private Bitmap InvaderImage(int animationCell) {
  // This is mostly a convenience method, and
  //   returns the right bitmap for the specified cell
}
```

每个入侵者知道自己的类型。所以如果为InvaderImage()方法提供了一个数（表示动画单元），它会返回一个Bitmap，其中包含适当的图片。

**记住，可以从以下地址下载这些图片：**

**http://www.headfirstlabs.com/hfcsharp/**

# 玩家的战舰可以移动，也可能没命

PlayerShip类跟踪玩家的战舰。这与Invaders类很类似，但是更简单一些。

Location和Area属性与Invader类中的相应属性非常相似。

| PlayerShip |
|---|
| Location: Point<br>Area: Rectangle<br>Alive: bool |
| Draw(g: Graphics)<br>Move(direction: Direction) |

一发炮弹击中战舰时，游戏将这个战舰的Alive属性设置为false。然后游戏让入侵者停止移动，直到战舰将其Alive属性重置为true。

Draw()方法在适当的位置画出玩家的战舰，除非玩家已经死了，在这种情况下，它会绘制战舰被击中而摧毁的动画。

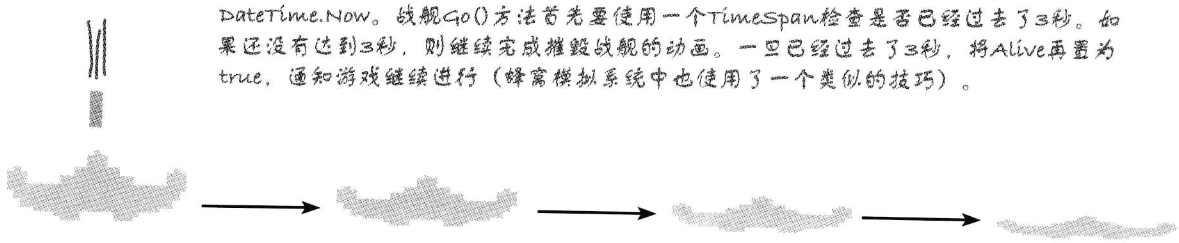
Move()方法取一个参数，即一个Direction enum，使玩家朝着这个方向移动。

PlayerShip的构造函数需要一个Rectangle参数，其中包含游戏的边界，Move()要确保战舰没有超出游戏的边界。

## 战舰被击中时完成动画

Draw()方法应当取一个Graphics对象作为参数。然后检查战舰的Alive属性。如果还活着，就使用其Location属性自行绘制。如果已经死了，不再在Graphics上绘制平常的位图，PlayerShip对象会使用其私有deadShipHeight字段完成玩家战舰的动画，缓慢地被炮弹摧毁。死后3秒，再将战舰的Alive属性重置回true。

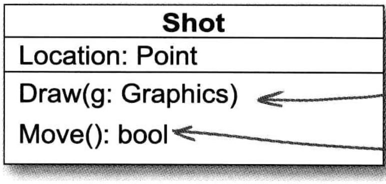
等待3秒很容易，只需使用Alive属性的设置存取方法将一个私有DateTime字段设置为DateTime.Now。战舰Go()方法首先要使用一个TimeSpan检查是否已经过去了3秒。如果还没有达到3秒，则继续完成摧毁战舰的动画。一旦已经过去了3秒，将Alive再置为true，通知游戏继续进行（蜂窝模拟系统中也使用了一个类似的技巧）。



```
public void Draw(Graphics g) {
  if (!Alive) {
        重置deadShipHeight字段，并绘制战舰。
    } else {
        检查deadShipHeight字段。如果大于0，则减1。
        使用DrawImage()把战舰画得扁一点。
    }
}
```

# “开火!”

Game有两个Shot对象列表：一个是玩家的炮弹，在屏幕上向上移动，另一个是敌人的炮弹，在屏幕上向下移。要完成Shot的工作，只需一个Point位置，一个绘制炮弹的方法，以及一个移动炮弹的方法。下面是它的类图：




Draw()完成炮弹的绘制，会为这个炮弹画出一个小矩形。每次更新屏幕时Game会调用这个方法。

Move()将炮弹上移或下移，并跟踪炮弹是否在游戏的边界以内。

以下代码可以作为Shot类的起点：

```
class Shot {
  private const int moveInterval = 20;
  private const int width = 5;
  private const int height = 15;

  public Point Location { get; private set; }

  private Direction direction;
  private Rectangle boundaries;

  public Shot(Point location, Direction direction,
            Rectangle boundaries) {
    this.Location = location;
    this.direction = direction;
    this.boundaries = boundaries;
  }

  // Your code goes here
}
```

可以调整这些常量增加或降低游戏的难度……炮弹越小就越容易躲开，速度越快则越难躲避。

炮弹在Move()方法中更新自己的位置，所以Location是一个只读自动属性。

Direction是一个enum，定义了Up和Down。

游戏将窗体的显示矩形作为boundaries参数传入Shot的构造函数，所以炮弹可以确定是否飞出了屏幕。

你的任务是确保Draw()取一个Graphics对象参数，把炮弹绘制为一个黄色矩形。然后，Move()应当将炮弹上移或下移，如果炮弹仍在游戏边界范围以内，则返回true。

# 一闪一闪亮晶晶……由你来完成

再给你一个提示：创建一个只包含一个窗体、一个Game类和Stars类的工程。看看能不能画出一个黑色天空，上面闪烁着星星。有了这个宇宙的基础，就可以增加其他类和方法了。

最后要完成的类是Stars类。总共有300个星星，这个类会跟踪所有这些星星，每次调用`Twinkle()`时会让其中的5个星星出现，另外让5个星星消失。

不过，首先需要为每个星星创建一个struct：

```
private struct Star {
  public Point point;
  public Pen pen;

  public Star(Point point, Pen pen) {
    this.point = point;
    this.pen = pen;
  }
}
```

每个星星有一个point（表示位置）和一个pen（表示颜色）。

Star所做的就是保存这些数据……没有任何行为。

Stars类要维护一个`List<Star>`保存300个`Star struct`。需要为Stars建立一个构造函数，填充这个列表。这个构造函数取一个包含显示边界的`Rectangle`，另外还需要一个`Random`实例用来创建随机的`Point`将各个星星放在随机的位置上。

可以在Stars.cs中定义Star struct，因为只有Stars需要用到这个struct。

以下是Stars的类图，包含了需要的所有方法：

| Stars |
|---|
| |
| Draw(g: Graphics)<br>Twinkle(random: Random) |

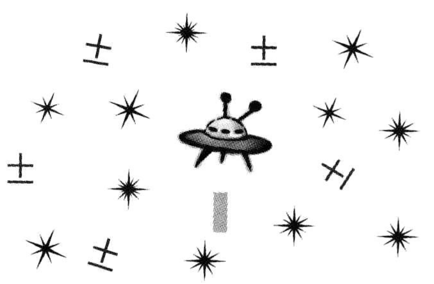
Draw()画出全部300颗星星……

……Twinkle()取出5个星星，并增加5个新的星星。

Game维护着一个Random实例，所有对象都可以使用这个实例。

`Draw()`应当画出列表中的所有星星，`Twinkle()`要删除5个随机的星星，并在它们的位置上增加5个新的星星。

可能还需要创建一个`RandomPen()`方法，这样每次创建一个新星星时可以很容易地得到一个随机的颜色。星星提供了5种颜色，这个方法会返回其中的一种颜色，它会生成0到4之间的一个数，并选择匹配的Pen对象。



# 还有很多可以做……

觉得这个游戏看起来还不错？只需再做一些补充，可以让它更上一层楼：

### 增加爆炸动画

击中入侵者后让入侵者爆炸，然后显示一个数，告诉玩家这个入侵者值多少分。

### 增加一个母舰

会有一个母舰（值250分）时不时地经过战场上方。如果玩家击中母舰，会得到额外奖励。

### 增加遮蔽

增加浮动的遮蔽，玩家可以藏在后面。遮蔽可以很简单，玩家和敌人的炮弹无法穿过。如果确实想让你的游戏很炫，不妨增加一些可以穿过的遮蔽，遮蔽如果被玩家和入侵者多次击中，就会在这个遮蔽上炸出洞来。

游戏中级别越高的关，打穿遮蔽所需的击中次数就越少。

### 增加俯冲轰炸机

创建一种特殊类型的敌人，可以俯冲轰炸玩家。发动俯冲轰炸的敌人不组成编队，它们朝着对手起飞，向下飞到屏幕最下方，然后恢复它的位置。

### 增加更多武器

开展一个军备竞赛！可以增加智能弹、激光武器、制导导弹…… 有各种武器可供玩家和入侵者相互攻击。看看能不能为这个游戏增加3个新武器。

好的类设计应当保证只需修改最少的代码就能改变图形化显示。

### 增加更多图形

可以到www.headfirstlabs.com/books/hfcsharp/ 为简单遮蔽、母舰等等找到更多图形文件。我们提供了一些粗糙的像素图形，看上去有一种80年代的风格。你能提供更多图形让游戏呈现一种新的风格吗？

**你可以借这个机会露一手！如果你为这个游戏开发了一个很酷的新版本，可以加入 Head First C#论坛（www.headfirstlabs.com/books/hfcsharp/）炫耀一番！**

附录i：其他

# 11 这本书最想介绍的10大内容




**好戏才刚刚开始！** 我们已经介绍了很多相当棒的工具，可以帮助你使用C#构建非常强大的软件。不过，这本书绝对无法涵盖每一个工具、技术或技巧，实在是篇幅不够。我们必须做出艰难的抉择，究竟包含哪些内容，而哪些内容无法涵盖。下面这一些内容就无法涵盖在内。不过尽管书中没有介绍，但是我们还是认为这些内容非常重要，也很有用，所以希望你们能有一些基本的认识。

我们真希望也能像介绍书中其他内容一样全面介绍这部分内容，不过实在是篇幅不够！但是我们还是希望为你提供一个很好的起点，能够由此了解更多信息。

# #1. 基础知识

开始之前，先给出这个附录中将一直用到的Guy类。先来看看这个类的注释。注意到了吗？这个类、它的方法以及属性都使用了三重斜线（///）注释。这称为XML注释，IDE会帮助你增加这些注释。只需在类、方法、属性或字段声明（以及很多其他地方）前面键入 “///”，IDE就会为它填写XML注释框架。之后，要使用这个属性、方法等等时，IDE会在它的智能提示窗口中显示XML注释中的信息。

类的XML注释包括一个<summary>块。注意它由一个<summary>开始，到</summary>结束。

```
/// <summary>
/// A guy with a name, age and a wallet full of bucks
/// </summary>

class Guy
{
    /*
     * Notice how Name and Age are properties with backing fields that are
     * marked readonly. That means those backing fields can only be set when
     * the object is initialized (in their declarations or in the constructor).
     */
    /// <summary>
    /// Read-only backing field for the Name property
    /// </summary>
    private readonly string name;

    /// <summary>
    /// The name of the guy
    /// </summary>
    public string Name { get { return name; } }

    /// <summary>
    /// Read-only backing field for the Name property
    /// </summary>
    private readonly int age;

    /// <summary>
    /// The guy's age
    /// </summary>
    public int Age { get { return age; } }

    /*
     * Cash is not readonly because it might change during the life of the Guy.
     */

    /// <summary>
    /// The number of bucks the guy has
    /// </summary>
    public int Cash { get; private set; }

    /// <summary>
```

合理地标志一个字段是一种很有用的封装工具，因为这说明一旦对象实例化，这个字段就不能再改变。

IDE为构造函数或其他方法增加骨架，它为各个参数增加<param>标记。

```
/// The constructor sets the name, age and cash
/// </summary>
/// <param name="name">The name of the guy</param>
/// <param name="age">The guy's age</param>
/// <param name="cash">The amount of cash the guy starts with</param>
public Guy(string name, int age, int cash) {
    this.name = name;
    this.age = age;
    Cash = cash;
}

public override string ToString() {
    return String.Format("{0} is {1} years old and has {2} bucks", Name, Age, Cash);
}

/// <summary>
/// Give cash from my wallet
/// </summary>
/// <param name="amount">The amount of cash to give</param>
/// <returns>The amount of cash I gave, or 0 if I don't have enough cash</returns>
public int GiveCash(int amount) {
    if (amount <= Cash && amount > 0)

    {
        Cash -= amount;
        return amount;
    }
    else
    {
        return 0;
    }
}

/// <summary>
/// Receive some cash into my wallet
/// </summary>
/// <param name="amount">Amount to receive</param>
/// <returns>The amount of cash received, or 0 if no cash was received</returns>
public int ReceiveCash(int amount) {
    if (amount > 0)
    {
        if (amount > 0)
        {
            Cash += amount;
            return amount;
        }
        Console.WriteLine("{0} says: {1} isn't an amount I'll take", Name, amount);
    }
    return 0;
}
}
```

在这里覆盖ToString()。这在第8章做过介绍。

## ……更多基础知识……

学习任何计算机语言时都很容易让人畏惧，C#也不例外。正是因为这个原因，我们主要强调（从我们的经验来看）对新手和中级开发人员来说最为常见的部分内容。不过还有一些基本的C#和.NET语法确实很有用，不过一旦熟悉这些语法，按你的进度来学习会容易得多。下面是一个控制台应用，展示了这样一些语法。

```
static void Main(string[] args)
{
    // We'll use these Guy and Random instances throughout this example.
    Guy bob = new Guy("Bob", 43, 100);
    Guy joe = new Guy("Joe", 41, 100);
    Random random = new Random();

   /*
    * Here are two useful keywords that you can use with loops. The "continue" keyword
    * tells the loop to jump to the next iteration of a loop, and the "break" keyword
    * tells the loop to end immediately.
    *
    * The break, continue, throw, and return statements are called "jump statements"
    * because they cause your program to jump to another place in the code when they're
    * executed. (You learned about break with switch/case statements in Chapter 8, and
    * the throw statement in Chapter 10.) There's one more jump statement, goto, which
    * jumps to a label. (You'll recognize these labels as having very similar syntax
    * to what you use in a case statement.)
    *
    * You could easily write this next loop without continue and break. That's a good
    * example of how C# lets you do the same thing many different ways. That's why you
    * don't need break, continue, or any of these other keywords or operators to write
    * any of the programs in this book.
    *
    * The break statement is also used with "case", which you can see in chapter 8.
    */
   while (true) {
       int amountToGive = random.Next(20);

       // The continue keyword jumps to the next iteration of a loop
       // Use the continue keyword to only give Joe amounts over 10 bucks
       if (amountToGive < 10)
           continue;

       // The break keyword terminates a loop early
       if (joe.ReceiveCash(bob.GiveCash(amountToGive)) == 0)
           break;

       Console.WriteLine("Bob gave Joe {0} bucks, Joe has {1} bucks, Bob has {2} bucks",
           amountToGive, joe.Cash, bob.Cash);
   }
   Console.WriteLine("Bob's left with {0} bucks", bob.Cash);
```

要掌握这个内容，一种很好的办法就是具体调试，使用监视来查看发生了什么。学习这本书时，就是在不断通过试验来了解这些概念。

很多人都认为jump语句是不好的实践。通常可以利用其他途径得到同样的结果。不过万一遇到jump语句，如果能知道它是如何工作的会很有用。

continue语句导致程序跳过迭代的其余代码，回到循环最前面。

break语句导致循环结束，程序转向Console.WriteLine()语句。

```
// The ?: conditional operator is an if/then/else collapsed into a single expression
// [boolean test] ? [statement to execute if true] : [statement to execute if false]
Console.WriteLine("Bob {0} more cash than Joe",
    bob.Cash > joe.Cash ? "has" : "does not have");

// The ?? null coalescing operator checks if a value is null, and either returns
// that value if it's not null, or the value you specify if it is
// [value to test] ?? [value to return if it's null]
bob = null;
Console.WriteLine("Result of ?? is '{0}'", bob ?? joe);
```

由于bob为null，??操作符会返回joe。

```
// Here's a loop that uses goto statements and labels. It's rare to see them, but
// they can be useful with nested loops. (The break statement only breaks out of
// the innermost loop)
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 3; j++)
    {
        if (i > 3)
            goto afterLoop;
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
afterLoop:
```

goto语句导致直接执行到一个标签。

标签是由字母、数字或下划线组成的字符串，后面跟着一个冒号。

```
// When you use the = operator to make an assignment, it returns a value that you
// can turn around and use in an assignment or an if statement
int a;
int b = (a = 3 * 5);
Console.WriteLine("a = {0}; b = {1};", a, b);
```

这个语句首先设置a和3*5，然后设置b为结果。

```
// When you put the ++ operator before a variable, it increments the variable
// first, and then executes the rest of the statement.
a = ++b * 10;
Console.WriteLine("a = {0}; b = {1};", a, b);
```

++b表示b首先自增1，然后a设置为b*10。

```
// Putting it after the variable executes the statement first and then increments
a = b++ * 10;
Console.WriteLine("a = {0}; b = {1};", a, b);
```

b++表示首先将a设置为b*10，然后b自增1。

```
/*
 * When you use && and || to do logical tests, they "short-circuit" -- which means
 * that as soon as the test fails, they stop executing. When (A || B) is being
 * evaluated, if A is true then (A || B) will always be true no matter what B is.
 * And when (A && B) is being evaluated, then if A is false then (A && B) will always
 * be false no matter what B is. In both of those cases, B will never get executed
 * because the operator doesn't need its value in order to come up with a return value.
 */

int x = 0;
int y = 10;
int z = 20;
```

我们会在下一页的代码中使用这些值！

使用/*和*/增加注释时，并不要求在各行最前面增加一个*，不过这样可以更易于阅读。

使用逻辑"与"和逻辑"或"操作符来"短路"属性也是一种很好的方法，可以用来有效地编写if/else语句。这就像是说"只有当(y < z)为true时才执行 (y / x == 4)。

```
// y / x will throw a DivideByZeroException because x is 0. But since (y < z) is true,
// the || operator knows it will be true without ever having to execute the other
// statement, so it short-circuits and never executes (y / x == 4)
if ((y < z) || (y / x == 4))
    Console.WriteLine( "this line printed because || short-circuited" );

// Since (y > z) is false, the && operator knows it will return false without
// executing the other statement, so it short-circuits and doesn't throw the exception
if ((y > z) && (y / x == 4))
    Console.WriteLine("this line will never print because && short-circuited");


/*
 * A lot of us think of 1's and 0's when we think of programming, and manipulating
 * those 1's and 0's is what logic operators are all about.
 */

// Use Convert.ToString() and Convert.ToInt32() to convert a number to or from a
// string of 1's and 0's in its binary form. The second argument specifies that you're
// converting to base 2.
string binaryValue = Convert.ToString(217, 2);
int intValue = Convert.ToInt32(binaryValue, 2);
Console.WriteLine("Binary {0} is integer {1}", binaryValue, intValue);

// The &, |, ^, and ~ operators are logical AND, OR, XOR, and bitwise complement
int val1 = Convert.ToInt32("100000001", 2);
int val2 = Convert.ToInt32("001010100", 2);
int or = val1 | val2;
int and = val1 & val2;
int xor = val1 ^ val2;
int not = ~val1;
```

所有整数类型、所有枚举和布尔类型都内置有&、|和^操作符。对于布尔类型来说，&和&&之间（以及|和||之间）唯一的区别是这些不会短路。

~是整数类型和枚举类型的逻辑非操作符，从某种意义上讲，这就类似于布尔类型的!操作符。

```
// Print the values -- and use the String.PadLeft() method to add leading 0's
Console.WriteLine("val1: {0}", Convert.ToString(val1, 2));
Console.WriteLine("val2: {0}", Convert.ToString(val2, 2).PadLeft(9, '0'));
Console.WriteLine("  or: {0}", Convert.ToString(or, 2).PadLeft(9, '0'));
Console.WriteLine(" and: {0}", Convert.ToString(and, 2).PadLeft(9, '0'));
Console.WriteLine(" xor: {0}", Convert.ToString(xor, 2).PadLeft(9, '0'));
Console.WriteLine(" not: {0}", Convert.ToString(not, 2).PadLeft(9, '0'));
// Notice what the ~ operator returned:  11111111111111111111111011111110
// It's the 32-bit complement of val1: 00000000000000000000000100000001
// The logical operators are operating on int, which is a 32-bit integer.
```

Convert.ToString()返回一个String对象，我们调用这个对象的PadLeft()方法用0来填充结果。

这在运行程序并查看输出时会很有意义。要记住，你并不需要键入所有这些代码，它们都可以从Head First Labs网站下载（http://www.headfirstlabs.com/books/hfcsharp）！

```
// The << and >> operators shift bits left and right. And you can combine any
// logical operator with =, so >>= or &= is just like += or *=.
int bits = Convert.ToInt32("11", 2);
for (int i = 0; i < 5; i++)
{
    bits <<= 2;
    Console.WriteLine(Convert.ToString(bits, 2).PadLeft(12, '0'));
}
for (int i = 0; i < 5; i++)
{
    bits >>= 2;
    Console.WriteLine(Convert.ToString(bits, 2).PadLeft(12, '0'));
}
```

这与逻辑没有任何关系，这只是你经常见到的一种有用的做法。

```
// You can instantiate a new object and call a method on it without
// using a variable to refer to it.
Console.WriteLine(new Guy("Harry", 47, 376).ToString());
```

```
// We've used the + operator for string concatenation throughout the book, and that
// works just fine. However, a lot of people avoid using + in loops that will have
// to execute many times over time, because each time + executes it creates an extra
// object on the heap that will need to be garbage collected later. That's why .NET
// has a class called StringBuilder, which is great for efficiently creating and
// concatenating strings together. Its Append() method adds a string onto the end,
// AppendFormat() appends a formatted string (using {0} and {1} just like
// String.Format() and Console.WriteLine() do), and AppendLine() adds a string
// with a line break at the end. To get the final concatenated string, call
// its ToString() method.
StringBuilder stringBuilder = new StringBuilder("Hi ");
stringBuilder.Append("there, ");
stringBuilder.AppendFormat("{0} year old guy named {1}. ", joe.Age, joe.Name);
stringBuilder.AppendLine("Nice weather we're having.");
Console.WriteLine(stringBuilder.ToString());

Console.ReadKey();
```

这里要注意一点：在这个特别的例子中，StringBuilder的表现会比+差，因为+会预计算串长，并准确得出要分配多少内存。

不能提前知道希望完成的连接数时，通常会使用StringBuilder。

```
/*
 * This is a good start, but it's by no means complete. Luckily, Microsoft gives you
 * a reference that has a complete list of all of the C# operators, keywords, and
 * other features of the language. Take a look through it -- and if you're just getting
 * started with C#, don't worry if it seems a little difficult to understand. MSDN
 * is a great source of information, but it's meant to be a reference, not a learning
 * or teaching guide.
 *
 * C# Programmer Reference: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx
 * C# Operators: http://msdn.microsoft.com/en-us/library/6a71f45d.aspx
 * C# Keywords: http://msdn.microsoft.com/en-us/library/x53a06bb.aspx
 */
}
```

# #2. 命名空间和程序集

我们决定这本书主要关注构建和运行应用时需要知道的真正实用的东西。每一章中，我们都在Visual Studio中创建工程，并在调试工具中运行。我们展示了编译代码最终会放在一个可执行文件中，并且向你展示了如何发布这个可执行文件，使其他人可以把它安装在他们自己的机器上。这对于完成这本书中的所有练习已经足够了，不过还有必要退一步，更仔细地了解你构建的究竟是什么。

编译C#程序时，就是在创建一个程序集。程序集是一个包含编译代码的文件。有两种类型的程序集。可执行文件（有时称为“处理程序集”）有EXE文件扩展名。这本书中编写的所有程序都编译为可执行文件。这些是可以执行的程序集（应该知道，这就是可以双击运行的EXE文件）。还有一种库程序集，它们有DLL文件扩展名。这些程序集包含可以在你的程序中使用的类，另外稍后就会看到，命名空间对于如何使用类起着很重要的作用。

要掌握程序集的基本知识，可以首先创建一个类库，然后构建一个使用这个类库的程序。先在Visual Studio中创建一个新的Class Library工程，名为Headfirst.Csharp.Leftover2。库最初创建时，其中包含文件Class.cs。删除这个文件，增加一个新的类，名为Guy.cs。打开这个新的Guy.cs文件：

```
namespace Headfirst.Csharp.Leftover2
{
    class Guy
    {
    }
}
```

注意到了吗？Visual Studio会使命名空间与类库名一致。这是一个非常标准的模式。

继续在Guy类中填入附录“其他”第1项中的代码，稍后我们就会使用这个类。接下来，再增加两个类，分别名为HiThereWriter和LineWriter。以下是HiThereWriter的代码：

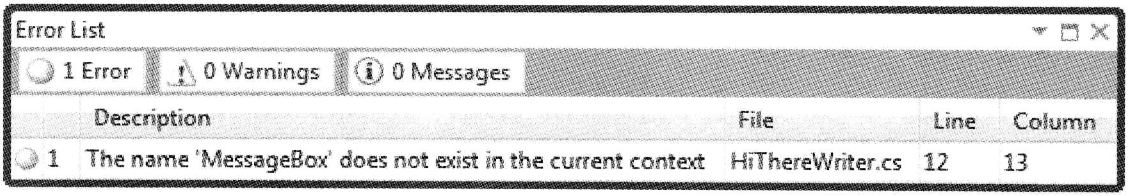
```
namespace Headfirst.Csharp.Leftover2
{
    public static class HiThereWriter
    {
      public static void HiThere(string name)
       {
          MessageBox.Show("Hi there! My name is " + name);
       }
    }
}
```

以下是LineWriter的代码（也放在Headfirst.Csharp.Leftover2命名空间中）：

```
    internal static class LineWriter {
        public static void WriteALine(string message)
        {
            Console.WriteLine(message);
        }
    }
```

> 我们把类库命名为Headfirst.Csharp.Leftover2，这是因为这是一种相当标准的命名程序集的方式。关于程序集命名的更多内容可以参考：
> http://msdn.microsoft.com/en-us/library/ms229048.aspx

现在试着编译你的程序。你会得到一个错误：



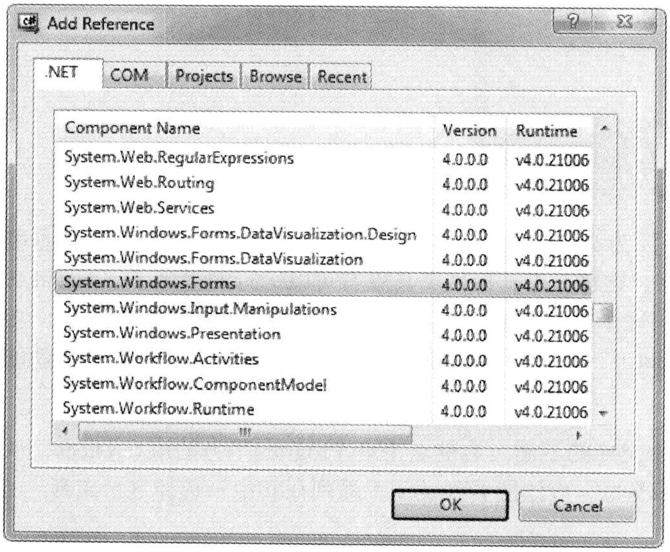

OK，没关系。我们知道如何修正这个错误。在类的最前面增加一行代码：

```
using System.Windows.Forms;
```

等一下，还是不能编译！这里有些奇怪。键入这行代码时，注意到没有，当你键入到“using System.Win”时，智能提示窗口不再提供建议。这是因为你的工程还没有引用**System.Windows.Forms**程序集。

下面来修正这个问题：引用正确的程序集。进入Solution Explorer，展开工程的“References”文件夹。右键点击这个文件夹，选择“Add Reference…”；应该会弹出一个窗口：

在.NET页上，开始键入“System.Windows.Forms”，它应当跳至这个程序集。确保突出显示这个程序集，并点击OK。现在System.Windows.Forms应当出现在Solution Explorer中的References文件夹下—而且你的程序能成功编译了！




这个窗口显示了你的程序可以访问的程序集。其中一些存储在全局程序集缓存（Global Assembly Cache，GAC）中，但是并不是GAC中的每一个程序集都会在这个窗口中出现。GAC是计算机上所有.NET程序都能访问的集中的程序集集合。如果在“开始”菜单（或者老版本Windows上为“开始/运行”）中键入%systemroot%\assembly，就可以看到其中的所有程序集。

现在花点时间做这个尝试。注意一些程序集可能有多个不同版本。你的程序可以引用一个特定的程序集版本，所以即使计算机上安装了一个更新的不兼容的版本，程序也不会出问题。

“Add References”窗口通过检查一个注册键而不是GAC来得出显示哪些程序集。有关的更多信息请访问：http://support.microsoft.com/kb/306149

## ……刚刚做了什么？

仔细查看LineWriter和HiThereWriter的声明：

```
public class HiThereWriter

internal static class LineWriter
```

类声明中有一些访问修饰符：`HiThereWriter`声明有public访问修饰符，`LineWriter`声明有internal访问修饰符。稍后，你会编写一个引用这个类库的控制台应用。一个程序只能直接访问另一个类库的公共类—不过这些类也可能间接访问，比如一个方法调用另一个方法，或者返回一个实现了某个公共接口的内部对象的实例。

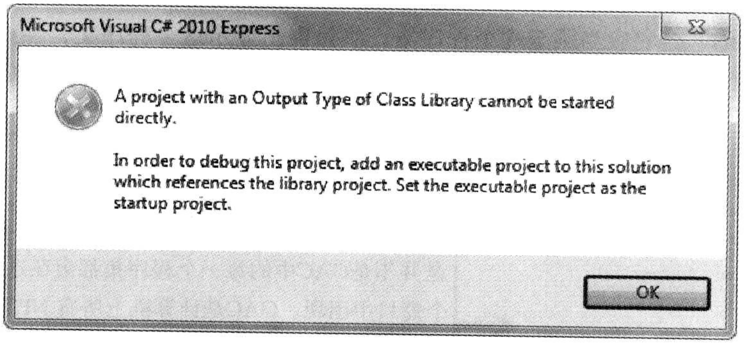
再来看我们的Guy类，查看它的声明：

```
class Guy
```

由于这里没有访问修饰符，所以默认为internal。我们希望从另一个类声明Guy，所以要把声明改为public：

```
public class Guy
```

接下来，试着在调试工具中运行这个程序。你会看到以下错误：




如果仔细考虑一下，你会发现这是有道理的，因为类库并没有入口点。类库只是可以由其他程序使用的一组类。所以下面增加一个使用这些类的可执行程序。这样一来，调试工具就有可以运行的程序了。Visual Studio有一个非常有用的特性，接下来我们就会充分利用这个特性：它可以在一个解决方案中加载多个工程。右键点击Solution Explorer，选择“Add”→“New Project…”，打开我们常见的Add Project窗口。增加一个新的控制台程序，名为MyProgram。

一旦增加新程序，它会出现在Solution Explorer中类库的下面。右键点击MyProgram下方的References，从菜单选择“Add reference...”。这一次会打开Projects标签页。应该能看到类库工程列在其中—选择这个工程，点击OK。现在它应当出现在References窗口中。

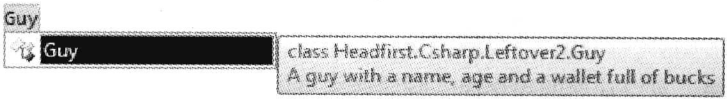
在新工程的Program.cs文件最上面增加下面这行using代码：

```
using Headfirst.Csharp.Leftover2;
```

注意在你键入时智能提示窗口会给出“Csharp”和“Leftover2”。

现在我们可以编写一个新程序。首先键入Guy。看看会弹出什么：

```
static void Main(string[] args)
{
```




智能提示窗口会列出Guy的完整的命名空间，所以你可以看到确实在使用另一个程序集中定义的类。完成这个程序：

```
static void Main(string[] args)
{
    Guy guy = new Guy("Joe", 43, 125);
    HiThereWriter.HiThere(guy.Name);
}
```

现在运行你的程序。噢，等一下，又会像前面一样得到同样的错误消息，因为类库是不能运行的！没问题，在Solution Explorer中右键点击这个新的MyProgram工程，选择“Set as Startup Project”。解决方案可以有多个不同的工程，我们就是以这种方式告诉它在调试工具中运行时应当启动哪一个工程。现在再次运行你的程序，这一次它能运行了！

本书中我们一直在告诉你编译你的代码。编译代码时，会编译为通用中间语言（Intermediate Language，IL），这是.NET使用的低级语言。这是一种人可读的汇编语言，所有.NET语言（包括C#和Visual Basic）都要编译为IL语言。使用CLR的即时编译器运行你的程序时，IL代码会编译为本地机器语言，即时编译器之所以得名就是因为它会在即将执行IL代码时将IL编译为本地代码（而不是在运行之前进行预编译）。

这说明你的EXE和DLL包含的都是IL，而不是本地汇编代码，这一点很重要，因为这说明很多语言可以编译为CLR能够运行的IL，包括Visual Basic .NET、F#、J#、managed C++/CLI、JScript .NET、Windows PowerShell、IronPython、Iron Ruby等等。这确实很有用：由于VB.NET代码编译为IL，所以可以在C#中构建一个程序集，而在一个VB.NET程序中使用（或者反之，在C#程序中使用VB.NET程序集）。

如果你的机器是Macintosh或Linux机器，可以尝试安装Mono。这是IL的一个开源实现，能够运行你在PC上构建的EXE文件（通常要键入“mono MyProgram.exe”，不过这只适用于一些.NET程序集）。不过，我们不打算过多地讨论这个内容，因为这本书主要关注Microsoft技术。但是我们必须承认，能够看到在Mac或Linux上本地运行Go Fish游戏或蜂巢模拟系统确实很酷！

## 为什么在第13章为类声明增加public

第13章中，你改变了Renderer、World、Hive、Flower和Bee类的声明，增加了public访问修饰符。为什么要那么做？

试着从Renderer声明删除public。尝试构建程序时你会得到类似下面的错误消息：

Inconsistent accessibility: property type ‘Beehive_Simulator.Renderer’ is less accessible than property ‘Beehive_Simulator.HiveForm.Renderer’

具体是怎么回事呢？来看HiveForm的类声明：

```
public partial class HiveForm : Form
```

这个声明你已经见过多次，可能已经不再注意它了。不过再仔细看看，IDE向工程增加一个窗体时，它会自动增加public访问修饰符。不过Renderer类的声明没有访问修饰符，所以它默认为internal。如果试图向public HiveForm类增加一个类型为Renderer的公共属性，构建会失败。不过，由于Renderer不是public，这导致了不一致的访问性错误。

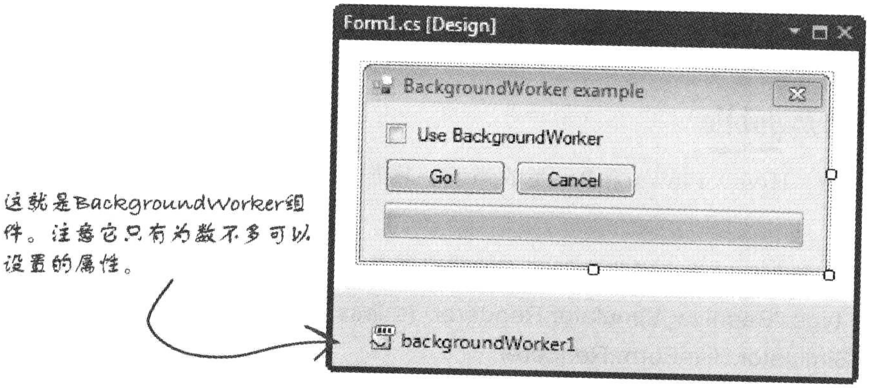
仔细想想这是有道理的。毕竟，程序也是一个程序集，另一个程序集可以访问它的类。如果另一个程序集试图引用HiveForm类会发生什么？它会查看类型Renderer的公共属性，不过由于Renderer类是内部的，它不能访问这个类。正是因为这个原因，有一个规则：如果你的程序集中有一个公共类，那么任何公共属性、方法或任何其他成员都只能使用公共类型。

**我们只是谈到了程序集的一点皮毛。还有很多相关内容（包括版本化和签名以提供安全性）。可以在这里了解更多有关程序集的内容：** http://msdn.microsoft.com/en-us/library/k3677y81.aspx

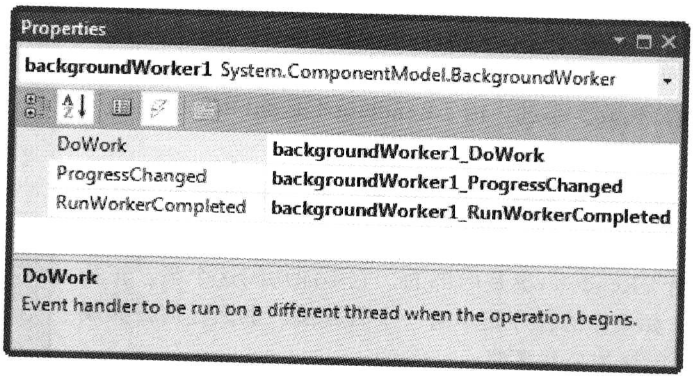
# #3. 使用BackgroundWorker让你的UI能够响应

在这本书中，我们已经向你介绍了两种方法，可以让你的程序同一时间做不只一件事。在第2章中，你已经了解如何使用Application.DoEvents()方法让窗体对按钮点击做出响应，而且仍然继续循环。不过，这不是一种很好的解决方案（原因有很多，我们没有一一解释），所以我们在第4章又给出了一种好得多的解决方案做不知一个定时器以固定的间隔触发一个事件。不过即使你知道如何使用定时器，有些情况下程序仍有可能很忙，变得无法响应。幸运的是，.NET提供了一个非常有用的组件，可以很容易地让你的程序在后台完成工作。这称为BackgroundWorker，下面将给出一个例子来说明它是如何工作的。

先来构建这个窗体。你要在这个窗体上拖入一个复选框（把它命名为useBackgroundWorkerCheckbox）、两个按钮（分别命名为goButton和cancelButton），还要拖入一个进度条（名为progressBar1）。然后向窗体拖入一个BackgroundWorker。它会出现在设计工具下方的灰色框中。保持名为backgroundWorker1不变，设置其WorkerReportsProgress和WorkerSupportsCancellation属性为true。




选择这个BackgroundWorker，进入属性窗口中的Events页（点击闪电图标）。它有3个事件：DoWork、ProgressChanged和RunWorkerCompleted。分别双击这3个事件，为各个事件增加一个事件处理程序。




窗体的代码在下面两页给出。

这是窗体的代码。

```
/// <summary>
/// Waste CPU cycles causing the program to slow down by doing calculations for 100ms
/// </summary>
private void WasteCPUCycles() {
    DateTime startTime = DateTime.Now;
    double value = Math.E;
    while (DateTime.Now < startTime.AddMilliseconds(100)) {
        value /= Math.PI;
        value *= Math.Sqrt(2);
    }
}
/// <summary>
/// Clicking the Go button starts wasting CPU cycles for 10 seconds
/// </summary>
private void goButton_Click(object sender, EventArgs e) {
    goButton.Enabled = false;
    if (!useBackgroundWorkerCheckbox.Checked) {
        // If we're not using the background worker, just start wasting CPU cycles
        for (int i = 1; i <= 100; i++) {
            WasteCPUCycles();
            progressBar1.Value = i;
        }
        goButton.Enabled = true;
    } else {
        cancelButton.Enabled = true;

        // If we are using the background worker, use its RunWorkerAsync()
        // to tell it to start its work
        backgroundWorker1.RunWorkerAsync(new Guy("Bob", 37, 146));
    }
}
/// <summary>
/// The BackgroundWorker object runs its DoWork event handler in the background
/// </summary>
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e) {
    // The e.Argument property returns the argument that was passed to RunWorkerAsync()
    Console.WriteLine("Background worker argument: " + (e.Argument ?? "null"));

    // Start wasting CPU cycles
    for (int i = 1; i <= 100; i++) {
        WasteCPUCycles();
        // Use the BackgroundWorker.ReportProgress method to report the % complete
        backgroundWorker1.ReportProgress(i);

        // If the BackgroundWorker.CancellationPending property is true, cancel
        if (backgroundWorker1.CancellationPending) {
            Console.WriteLine("Cancelled");
            break;
        }
    }
}
```

WasteCPUCycles()完成大量数学计算，从而占用CPU 100毫秒时间，然后返回。

用户点击Go!按钮时，事件处理程序会查看“Use BackgroundWorker”复选框是否选中。如果未选中，窗体会耗费CPU周期10秒时间。如果选中，完全符合本会调用BackgroundWorker的RunWorkerAsync()方法告诉它开始在后台完成工作。

如果窗体在使用后台工作进程，它会启用Cancel按钮。

告诉一个BackgroundWorker开始工作时，可以给它提供一个参数。在这里，我们为它传入了一个Guy对象（这个对象的定义见本附录前面的第1项）。

这是一个很好的例子，说明了如何使用我们在附录第1项中介绍的?? null接合操作符。如果e.Argument为null，这会返回“null”，否则它会返回e.Argument。

CancellationPending方法检查BackgroundWorker的CancelAsync()方法是否调用。

调用BackgroundWorker的RunWorkerAsync()方法时，它开始在后台运行其DoWork事件处理方法。注意它仍在调用同一个WasteCPUCycles()方法来耗费CPU周期。它还在调用ReportProgress()方法来报告一个完成百分数（0到100之间的一个数）。

BackgroundWorker只在WorkerReportsProgress和WorkerSupportsCancellation属性设置为true时触发其ProgressChanged和RunWorkerCompleted事件。

```
/// <summary>
/// BackgroundWorker fires its ProgressChanged event when the worker thread reports progress
/// </summary>
private void backgroundWorker1_ProgressChanged(object sender, ProgressChangedEventArgs e) {
    progressBar1.Value = e.ProgressPercentage;
}
```

DoWork事件处理程序调用ProgressChanged()方法时，它会导致BackgroundWorker产生ProgressChanged事件，并设置ProgressPercentage为传入的百分数。

```
/// <summary>
/// BackgroundWorker fires its RunWorkerCompleted event when its work is done (or cancelled)
/// </summary>
private void backgroundWorker1_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    goButton.Enabled = true;
    cancelButton.Enabled = false;
}
```

工作完成时，RunWorkerCompleted事件处理程序重新启用Go!按钮，并禁用Cancel按钮。

```
/// <summary>
/// When the user clicks Cancel, call BackgroundWorker.CancelAsync() to send it a cancel message
/// </summary>
private void cancelButton_Click(object sender, EventArgs e) {
    backgroundWorker1.CancelAsync();
}
```

如果用户点击Cancel，它调用BackgroundWorker的CancelAsync()方法来提供要取消的消息。

一旦窗体可以正常工作，运行这个程序。很容易看到BackgroundWorker可以让你的程序更具有响应性：

★ 确保“Use BackgroundWorker”复选框未选中，然后点击Go!按钮。你会看到进度条开始填充。试着来回拖动窗体—你会发现窗体无法拖动。窗体完全被锁定。如果幸运，最终对鼠标拖动事件做出响应时，窗体可能稍稍有所跳动。

★ 完成以上尝试后，选中“Use BackgroundWorker”复选框，并再次点击Go!按钮。这一次，窗体有很好的响应性。你可以到处移动窗体，甚至可以将它关闭，不会有任何延迟。完成时，它使用RunWorkerCompleted方法重新启用按钮。

★ 程序运行时（使用BackgroundWorker），点击Cancel按钮。这会更新其CancellationPending属性，会告诉程序取消，并退出循环。

你是不是奇怪为什么需要使用`ReportProgress()`方法而不是直接设置`ProgressBar`的`Value`属性？可以试试看。在DoWork事件处理程序中增加下面这行代码：

```
progressBar1.Value = 10;
```

然后再运行程序。一旦运行到这行代码，它会抛出一个InvalidOperationException，并给出如下消息：“Cross-thread operation not valid: Control ‘progressBar1’ accessed from a thread other than the thread it was created on.”（不合法的跨线程操作：访问控件‘progressBar1’的线程并非创建该控件的线程）。抛出这个异常的原因是，BackgroundWorker开始了另外一个线程，并在这个线程上执行DoWork方法。所以这里有两个线程：运行窗体的GUI线程和后台线程。.NET的一个线程规则是只有GUI线程能够更新窗体控件；否则，就会抛出这个异常。

这只是让新手困惑的众多线程陷阱之一，正是因为存在这么多问题，所以这本书中我们没有讨论线程。如果你想了解线程，强烈推荐Joe Albahari编写的一本有关C#和.NET中线程的非常棒的电子书：http://www.albahari.com/threading

# #4. Type类和GetType()

C#编程语言最强大的方面之一是它丰富的类型系统。不过在对构建程序有一定经验之前，将很难理解这个类型系统。实际上，最初看来这个类型系统实在让人费解。不过我们希望至少能让你对C#和.NET中类型如何工作有所认识。下面是一个控制台应用，可以向你简单介绍一些工具，你可以利用这些工具来处理类型。

```
class Program {
    class NestedClass {
        public class DoubleNestedClass {
            // Nested class contents ...
        }
    }

    static void Main(string[] args) {
        Type guyType = typeof(Guy);
        Console.WriteLine( "{0} extends {1}",
            guyType.FullName,
            guyType.BaseType.FullName);
        // output: TypeExamples.Guy extends System.Object

        Type nestedClassType = typeof(NestedClass.DoubleNestedClass);
        Console.WriteLine(nestedClassType.FullName);
        // output: TypeExamples.Program+NestedClass+DoubleNestedClass

        List<Guy> guyList = new List<Guy>();
        Console.WriteLine(guyList.GetType().Name);
        // output: List`1

        Dictionary<string, Guy> guyDictionary = new Dictionary<string, Guy>();
        Console.WriteLine(guyDictionary.GetType().Name);
        // output: Dictionary`2

        Type t = typeof(Program);
        Console.WriteLine(t.FullName);
        // output: TypeExamples.Program

        Type intType = typeof(int);
        Type int32Type = typeof(Int32);
        Console.WriteLine("{0} - {1}", intType.FullName, int32Type.FullName);
        // System.Int32 - System.Int32

        Console.WriteLine("{0} {1}", float.MinValue, float.MaxValue);
        // output:-3.402823E+38 3.402823E+38

        Console.WriteLine("{0} {1}", int.MinValue, int.MaxValue);
        // output:-2147483648 2147483647

        Console.WriteLine("{0} {1}", DateTime.MinValue, DateTime.MaxValue);
        // output: 1/1/0001 12:00:00 AM 12/31/9999 11:59:59 PM

        Console.WriteLine(12345.GetType().FullName);
        // output: System.Int32

        Console.ReadKey();
    }
}
```

这里只是简单提一句，不过要记住，类之间可以相互嵌套。Program包含NestedClass，而它又包含DoubleNestedClass。

这是入口点……

可以使用typeof关键字将一个类型（如Guy、int或DateTime）转换为一个Type对象。然后可以找出它的全名和基类型（如果没有继承任何类型，它的基类型就是System.Object）。

得到一个泛型的类型时，它的名字就是类型名后面加一个反引号，然后是其泛型参数的个数。

这是System.Type类。GetType()方法返回一个Type对象。

程序第一部分中使用的FullName属性是System.Type的一个成员。

float是System.Single的一个别名，int是System.Int32的别名。它们都是struct（有关内容在第14章介绍过）。

数字值类型和DateTime都有MinValue和MaxValue属性，会返回最小和最大合法值。

字面量也有类型！可以使用GetType()得到这些类型。

**关于类型有太多内容需要了解！可以从以下地址了解：**
**http://msdn.microsoft.com/en-us/library/ms173104.aspx**

# #5. 相等性、IEquatable和Equals()

本书中，想要比较两个变量中的值时，都会使用==操作符。不过你已经知道，一切都相等，只是一些值比另外一些“更相等”。==操作符只适用值类型（如int、double、DateTime或其他struct），但是用这个操作符比较引用类型时，只能比较两个引用变量是否指向同一个对象（或者是否都为null）。这本身并没有不对，不过C#和.NET提供了一组丰富的工具来处理对象中的值相等性。

首选，每个对象都有一个Equals()方法，默认地只有当传入它本身的一个引用时这个方法才会返回true。另外还有一个静态方法Object.ReferenceEquals()，它取两个参数，如果两个参数都指向同一个对象（或者都为null），这个方法就会返回true。下面给出一个例子，你可以在一个控制台应用中自己尝试一下：

```
Guy joe1 = new Guy("Joe", 37, 100);
Guy joe2 = joe1;
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));  // True
Console.WriteLine(joe1.Equals(joe2));                   // True
Console.WriteLine(Object.ReferenceEquals(null, null));  // True

joe2 = new Guy("Joe", 37, 100);
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));  // False
Console.WriteLine(joe1.Equals(joe2));                   // False
```

这里仍然使用本附录第1项中定义的Guy类。

但是这还只是开始。.NET内置有一个名为IEquatable<T>的接口，可以用来向对象增加代码，使它们能够区分是否与其他对象相等。实现了IEquatable<T>的对象知道如何将它的值与类型T的一个对象的值比较。它有一个方法：Equals()，要实现这个方法，编写代码将当前对象的值与另一个对象的值进行比较。在以下MSDN页上有更多有关信息(http://msdn.microsoft.com/en-us/library/ms131190.aspx)。下面摘录其中重要的一段文字：

如果不这样做，编译器会给出一个警告。

“如果实现Equals，还应当覆盖Object.Equals(Object)和GetHashCode的基类实现，使它们的行为与IEquatable<T>.Equals方法的行为一致。如果确实覆盖了Object.Equals(Object)，在你的类上调用静态Equals(System.Object, System.Object)方法时也会调用覆盖后的实现。这样可以确保Equals方法的所有调用都返回一致的结果，这正是这个例子所要展示的。”

这里是一个名为EquatableGuy的类，它扩展了Guy，并实现了IEquatable<Guy>：

```
/// <summary>
/// A guy that knows how to compare itself with other guys
/// </summary>
class EquatableGuy : Guy, IEquatable<Guy> {

    public EquatableGuy(string name, int age, int cash)
        : base(name, age, cash) { }

    /// <summary>
    /// Compare this object against another EquatableGuy
    /// </summary>
    /// <param name="other">The EquatableGuy object to compare with</param>
    /// <returns>True if the objects have the same values, false otherwise</returns>
    public bool Equals(Guy other) {
        if (ReferenceEquals(null, other)) return false;
        if (ReferenceEquals(this, other)) return true;
        return Equals(other.Name, Name) && other.Age == age && other.Cash == Cash;
    }
```

Equals()方法比较另一个Guy对象字段中的具体值，检查它的Name、Age和Cash，来查看它们是否相等，只有在相等时才返回true。

```
    /// <summary>
    /// Override the Equals method and have it call Equals(Guy)
    /// </summary>
    /// <param name="obj">The object to compare to</param>
    /// <returns>True if the value of the other object is equal to this one</returns>
    public override bool Equals(object obj) {
        if (!(obj is Guy)) return false;
        return Equals((Guy)obj);
    }
```

由于另一个Equals()方法已经完成了对Guy的比较，所以只需调用这个方法。

还要覆盖从Object继承的Equals()方法，以及GetHashCode（因为MSDN文章中提到的契约）

```
    /// <summary>
    /// Part of the contract for overriding Equals is that you need to override
    /// GetHashCode() as well. It should compare the values and return true
    /// if the values are equal.
    /// </summary>
    /// <returns></returns>
    public override int GetHashCode() {
        const int prime = 397;
        int result = age;
        result = (result * prime) ^ (Name != null ? Name.GetHashCode() : 0);
        result = (result * prime) ^ Cash;
        return result;
    }
}
```

这对于GetHashCode()是一个相当标准的模式。注意位异或操作符XOR（^）、质数和条件操作符(?:)的使用。

使用Equals()比较两个EquatableGuy对象时会是这样。

```
joe1 = new EquatableGuy("Joe", 37, 100);
joe2 = new EquatableGuy("Joe", 37, 100);
Console.WriteLine(Object.ReferenceEquals(joe1, joe2));   // False
Console.WriteLine(joe1.Equals(joe2));                    // True

joe1.GiveCash(50);
Console.WriteLine(joe1.Equals(joe2));                    // False
joe2.GiveCash(50);
Console.WriteLine(joe1.Equals(joe2));                    // True
```

Guy.Equals()只在对象的具体值相等时才返回true。

由于Equals()和GetHash(odel)都实现为检查字段和属性的值，所以List.(ontain)方法现在会返回true。这里有一个List<Guy>，其中包含多个Guy对象，还包括一个新的EquatableGuy对象，它与joel引用的对象有相同的值。

```
List<Guy> guys = new List<Guy>() {
    new Guy("Bob", 42, 125),
    new EquatableGuy(joe1.Name, joe1.Age, joe1.Cash),
    new Guy("Ed", 39, 95)
};

Console.WriteLine(guys.Contains(joe1));          // True

Console.WriteLine(joe1 == joe2);                 // False
```

List.Contains()会检查它的内容，调用各个对象的Equals()方法将它与传入的引用比较。

尽管joe1和joe2指向有相同值的对象，==和!=仍然只比较引用，而不是值本身。

对此我们能做点什么？翻到下一页，找出答案！

如果试图用==或!=操作符比较两个EquatableGuy引用，它们只是检查这两个引用是否指向同一个对象，或者是否都为null，不过，如果你希望具体比较对象的值该怎么做呢？实际上可以重载一个操作符,重新定义，从而在处理某种特定类型时完成一些特定的工作。下面可以看到一个例子，来说明如果对EquatableGuyWithOverload类进行比较，这个类扩展了EquatableGuy，并且增加了==和=!操作符的重载实现：

```
/// <summary>
/// A guy that knows how to compare itself with other guys
/// </summary>
class EquatableGuyWithOverload : EquatableGuy
{
    public EquatableGuyWithOverload(string name, int age, int cash)
        : base(name, age, cash) { }

    public static bool operator ==(EquatableGuyWithOverload left,
                                   EquatableGuyWithOverload right)
    {
        if (Object.ReferenceEquals(left, null)) return false;
        else return left.Equals(right);
    }

    public static bool operator !=(EquatableGuyWithOverload left,
                                   EquatableGuyWithOverload right)
    {
        return !(left == right);
    }

    public override bool Equals(object obj) {
        return base.Equals(obj);
    }

    public override int GetHashCode() {
        return base.GetHashCode();
    }
}
```

如果我们使用==来检查null而不是Object.ReferenceEquals()，会得到一个StackOverflowException异常。你能想出这是为什么吗？

由于我们已经定义了==，只需取反来完成!=的定义。

如果我们没有覆盖Equals()和GetHashCode()，IDE会给出这个警告：'EquatableGuyWithOverload' defines operator == or operator != but does not override Object.GetHashCode()（'EquatableGuyWithOverload'定义了操作符==或操作符!=，但是没有覆盖Object.GetHashCode()）。

由于EquatableGuyWithOverload可以作为EquatableGuy和Guy，所以我们可以直接基类方法。

Here's some code that uses EquatableGuyWithOverload objects:

```
joe1 = new EquatableGuyWithOverload(joe1.Name, joe1.Age, joe1.Cash);
joe2 = new EquatableGuyWithOverload(joe1.Name, joe1.Age, joe1.Cash);
Console.WriteLine(joe1 == joe2);    // False
Console.WriteLine(joe1 != joe2);    // True

Console.WriteLine((EquatableGuyWithOverload)joe1 ==
                               (EquatableGuyWithOverload)joe2);   // True
Console.WriteLine((EquatableGuyWithOverload)joe1 !=
                               (EquatableGuyWithOverload)joe2);   // False
joe2.ReceiveCash(25);
Console.WriteLine((EquatableGuyWithOverload)joe1 ==
                               (EquatableGuyWithOverload)joe2);   // False
Console.WriteLine((EquatableGuyWithOverload)joe1 !=
                               (EquatableGuyWithOverload)joe2);   // True
```

等一下，发生了什么，它在调用Guy的==和=!操作符。要强调转换为EquatableGuyWithOverload来调用正确的==和=!。

# #6. 使用yield return创建可枚举的对象

第8章我们了解了IEnumerable接口，以及如何在foreach循环中使用。C#和.NET提供了一些有用的工具来建立你自己的集合，首先要从IEnumerable接口开始。假设你希望创建你自己的枚举器，按顺序从这个Sport enum返回值：

```
enum Sport
{
    Football, Baseball,
    Basketball, Hockey,
    Boxing, Rugby, Fencing,
}
```

可以自行手动实现IEnumerable，建立Current属性和MoveNext()方法：

```
class SportCollection : IEnumerable<Sport> {
    public IEnumerator<Sport> GetEnumerator() {
        return new ManualSportEnumerator();
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    class ManualSportEnumerator : IEnumerator<Sport> {
        int current = -1;
        public Sport Current { get { return (Sport)current; } }
        public void Dispose() { return; } // Nothing to dispose
        object System.Collections.IEnumerator.Current { get { return Current; } }
        public bool MoveNext() {
            int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
            if ((int)current >= maxEnumValue)
                return false;
            current++;

            return true;
        }
        public void Reset() { current = 0; }
    }
}
```

IEnumerable只包含一个方法，即GetEnumerator()，不过我们还需要为它返回的枚举器建立类。

枚举器实现了IEnumerator<Sport>。foreach循环使用其Current属性和MoveNext()方法。

MoveNext()方法将current增1，并用它返回enum中的下一个运动。

下面是一个foreach循环，循环处理ManualSportCollection。它会按顺序返回运动（Football、Baseball、Basketball、Hockey、Boxing、Rugby和Fencing）：

```
Console.WriteLine( "SportCollection contents:" );
SportCollection sportCollection = new SportCollection();
foreach (Sport sport in sportCollection)
    Console.WriteLine(sport.ToString());
```

建立枚举器有很多工作，它必须管理它自己的状态，跟踪返回哪一个运动。幸运的是，C#提供了一个非常有用的工具，可以帮助你轻松地建立枚举器。这称为yield return，翻到下一页了解更多有关内容。

需要提醒第15章谈到的一点：所有集合都是可枚举的，但是并不是所有可枚举的对象在理论上都是集合，除非它实现了ICollection<T>接口。我们没人介绍如何从零开始建立集合，不过了解枚举器足以让你完成这个工作。

yield return语句是一种“一步到位的”自动枚举枚举器生成器。这个SportCollection类完成的工作与上一页的类是相同的，不过它的枚举器只有3行：

```
class SportCollection : IEnumerable<Sport> {
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    public IEnumerator<Sport> GetEnumerator() {
        int maxEnumValue = Enum.GetValues(typeof(Sport)).Length - 1;
        for (int i = 0; i < maxEnumValue; i++) {
            yield return (Sport)i;
        }
    }
}
```

就像前面说过的，这只是SportCollection类的起点。你可能还希望实现ICollection<Sport>接口。

看上去有点奇怪，不过如果具体调试，你就能看出会发生什么。编译器看到一个包含yield return语句的方法而且返回一个IEnumerator或IEnumerator<T>时，它会自动增加MoveNext()和Current方法。执行时，它遇到的第一个yield return会让它将第一个值返回到foreach循环。foreach循环继续时（通过调用MoveNext()方法），它会从执行的最后一个yield return后面的语句继续执行。如果枚举器方法返回，MoveNext()方法会返回false。如果在纸上读这些代码理解起来可能有些困难，不过如果把它加载调试工具，使用 Step Into (F11)单步跟踪，理解起来就会容易得多。为了更容易一些，这里给出一个非常简单的枚举器，名为NameEnumerator()，它会迭代处理4个名字：

```
static IEnumerable<string> NameCollection() {
    yield return "Bob";   // The method exits after this statement ...
    yield return "Harry"; // ... and resumes here the next time through
    yield return "Joe";
    yield return "Frank";
}
```

下面是一个迭代处理这个集合的foreach循环。使用Step Into (F11)查看到底发生了什么：

```
IEnumerable<string> names = NameEnumerator(); // Put a breakpoint here
foreach (string name in names)
    Console.WriteLine(name);
```

集合中通常还会看到索引器（indexer）。使用中括号[]获取一个列表、数组或字典中的一个对象时（如myList[3]或myDictionary[“Steve”]），就是在使用索引器。索引器实际上就是一个方法。它看起来非常类似于属性，只不过它有一个命名参数。

IDE有一个非常有用的代码段。键入索引器然后按下两个制表键（tab），IDE会为你自动增加索引器的骨架代码。

下面是SportCollection类的索引器：

```
public Sport this[int index] {
    get { return (Sport)index; }
}
```

为这个索引器传入3，会返回enum值Hockey。

下面是一个IEnumerable<Guy>，它跟踪了一群人，并提供一个索引器，使你能够获取或设置这些人的年龄。

```
class GuyCollection : IEnumerable<Guy> {
  private static readonly Dictionary<string, int> namesAndAges = new Dictionary<string, int>()
  {
          {"Joe", 41}, {"Bob", 43}, {"Ed", 39}, {"Larry", 44}, {"Fred", 45}
  };
```

枚举器使用这个私有字典来记录它创建的人，但是在使用其枚举器之前它并不具体创建Guy对象本身。

```
  public IEnumerator<Guy> GetEnumerator() {
      Random random = new Random();
      int pileOfCash = 125 * namesAndAges.Count;

      int count = 0;
      foreach (string name in namesAndAges.Keys) {
          int cashForGuy = (++count < namesAndAges.Count) ? random.Next(125) : pileOfCash;
          pileOfCash -= cashForGuy;
          yield return new Guy(name, namesAndAges[name], cashForGuy);
      }
  }
```

它创建现金数额随机的Guy对象。这样做只是为了展示枚举器可以在一个foreach循环中动态创建对象。

```
  System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {
      return GetEnumerator();
  }

  /// <summary>
  /// Gets or sets the age of a given guy
  /// </summary>
  /// <param name="name">Name of the guy</param>
  /// <returns>Age of the guy</returns>
  public int this[string name] {
      get {
          if (namesAndAges.ContainsKey(name))
              return namesAndAges[name];
          throw new IndexOutOfRangeException("Name " + name + " was not found");
      }
```

向索引器传入一个非法的索引时，通常会抛出一个IndexOutOfRangeException异常。

```
      set {
          if (namesAndAges.ContainsKey(name))
              namesAndAges[name] = value;
          else
              namesAndAges.Add(name, value);
      }
  }
}
```

这个索引器有一个设置存取方法，可能更新一个人的年龄，或者向字典增加新的人。

这里的代码使用索引器来更新一个人的年龄，并另外增加两个人，然后循环处理这些人：

```
Console.WriteLine("Adding two guys and modifying one guy");
guyCollection["Bob"] = guyCollection["Joe"] + 3;
guyCollection["Bill"] = 57;
guyCollection["Harry"] = 31;
foreach (Guy guy in guyCollection)
   Console.WriteLine(guy.ToString());
```

# #7. 重构

重构（refactoring）是指调整代码的结构而不改变它的行为。编写复杂的方法时，就应该花些时间仔细检查，确定能不能通过调整代码结构使它更易于理解。幸运的是，IDE已经内置了一些很有用的重构工具。我们可以完成多种重构，下面是经常使用的几种。

## 抽取方法

为第13章编写基于控件的Renderer时，原来使用了以下foreach循环：

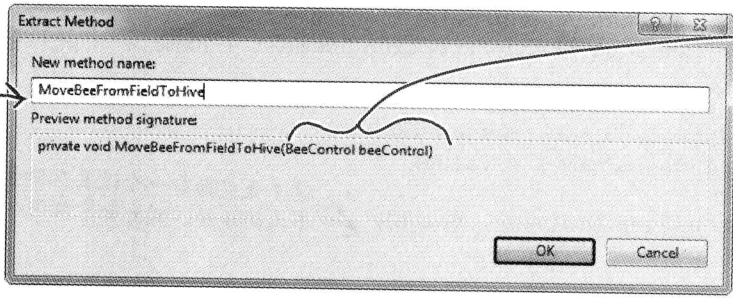
```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl)) {
            fieldForm.Controls.Remove(beeControl);
            beeControl.Size = new Size(40, 40);
            hiveForm.Controls.Add(beeControl);
            beeControl.BringToFront();
    } else if (hiveForm.Controls.Contains(beeControl)) {
        hiveForm.Controls.Remove(beeControl);
        beeControl.Size = new Size(20, 20);
        fieldForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    beeControl.Location = bee.Location;
}
```

这4行代码将一个BeeControl从Field窗体移到Hive窗体。

这4行代码将一个BeeControl从Hive窗体移到Field窗体。

我们的技术审校之一Joe Albahari指出这个代码可读性不好。他建议把这两个包含4行代码的代码块抽取到方法中。所以我们选择第一个代码块，右键点击并选择“Refactor”→“Extract Method...”。这会弹出以下窗口：

为这个新方法键入一个名字。我们将它命名为MoveBeeFromFieldToHive()因为这个名字能很好地描述这段代码要做什么。




IDE检查我们选择的代码，并确定它使用了一个名为beeControl的BeeControl变量，因此把它增加为这个方法的一个参数。

然后对另外一个包含4行代码的代码块完成同样的处理，把它抽取到一个名为MoveBeeFromHiveToField()的方法。最后foreach循环会变成以下形式，现在代码读起来就容易多了：

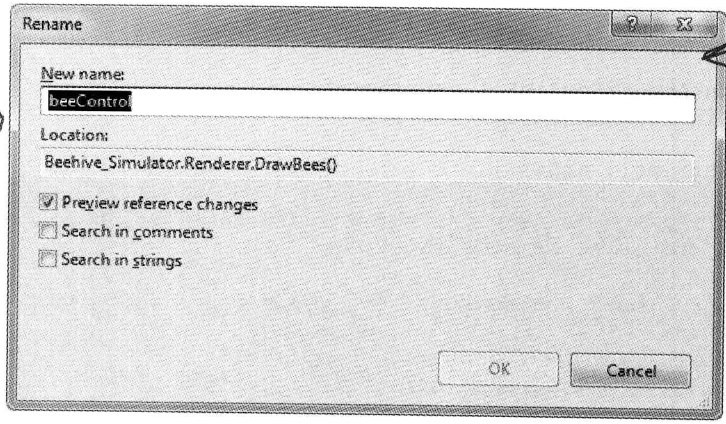
```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl))
            MoveBeeFromFieldToHive(beeControl);
    } else if (hiveForm.Controls.Contains(beeControl))
        MoveBeeFromHiveToField(beeControl, bee);
    beeControl.Location = bee.Location;
}
```

## 重命名变量

再来看第3章，我们解释了为类、方法、字段和变量选择直观的名字可以使代码更易于理解。在代码中为变量等命名时，IDE也能提供很大帮助。只需右键点击类、变量、字段、属性、命名空间或常量（可以是能命名的任何内容），并选择“Refactor >> Rename”。也可以直接使用F2，这会很方便，因为一旦开始重命名，你会发现往往一直都要重命名。

下面选择蜂巢模拟系统代码中的“beeControl”对它重命名，会弹出以下窗口：

利用这个窗口可以为这一项选择一个新名字。如果将它重命名，例如命名为“Bobbo”，IDE会检查代码，把这一项的每一个出现都替换为“Bobbo”。




IDE会全面完成重命名工作。如果对一个类重命名，它会修改实例化这个类或使用了这个类的每一条语句。可以在代码中的任何位置点击这个名字的任意一个出现，IDE会对程序中这个名字的所有出现进行修改。

## 合并条件表达式

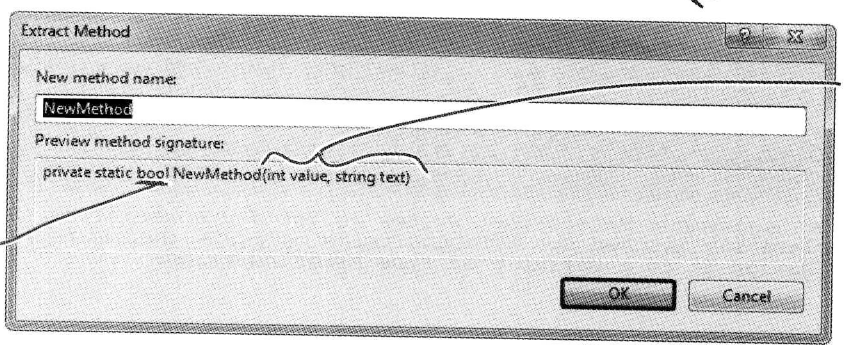
下面是使用“抽取方法”特性的一种简便方法。打开任意一个程序，增加一个按钮，再将以下代码增加到它的事件处理程序：

```
private void button1_Click(object sender, EventArgs e) {
    int value = 5;
    string text = "Hi there";
    if (value == 36 || text.Contains("there"))
        MessageBox.Show("Pow!");
}
```

选择if语句中的所有内容： value == 36 || text.Contains（“there”）。然后右键点击，并选择“Refactor >> Extract Method...”，会弹出以下窗口：

另外，它甚至会发现应当创建一个静态方法，因为这里没有使用任何字段。

每个条件表达式计算为一个bool，所以IDE会创建一个方法，它会返回一个bool，并把条件测试替换为该方法的一个调用。




这个表达式使用了两个变量，分别名为value和text，所以IDE使用这两个名字为方法增加参数。

这样不仅能让代码更易读，而且现在有了一个新方法，可以在别的地方重用。

# #8. 匿名类型、匿名方法和lambda表达式

C#允许创建类型和方法，而不需要使用显式命名的声明。如果类型或方法声明时没有命名就称为匿名类型或匿名方法。这些工具非常强大。例如，如果没有这些工具根本不可能有LINQ。不过一旦熟练掌握这种语言，掌握匿名类型、匿名方法和lambda表达式就会容易得多，所以本书中没有包括这些内容。不过这里可以做一个简要的介绍，以便你有所了解。

```
class Program {
    delegate void MyIntAndString(int i, string s);
    delegate int CombineTwoInts(int x, int y);

    static void Main(string[] args) {
        /*
         * In Chapter 15, you saw how the var keyword let the IDE determine the
         * type of an object at compile time.
         *
         * You can also create objects with anonymous types using var and new.
         *
         * You can learn more about anonymous types here:
         * http://msdn.microsoft.com/en-us/library/bb397696.aspx
         */

        // Create an anonymous type that looks a lot like a guy:
        var anonymousGuy = new { Name = "Bob", Age = 43, Cash = 137 };

        // When you type this in, the IDE's IntelliSense automatically picks up
        // the members -- Name, Age and Cash show up in the IntelliSense window.
        Console.WriteLine("{0} is {1} years old and has {2} bucks",
            anonymousGuy.Name, anonymousGuy.Age, anonymousGuy.Cash);
        // Output: Bob is 43 years old and has 137 bucks

        // An instance of an anonymous type has a sensible ToString() method.
        Console.WriteLine(anonymousGuy.ToString());
        // Output: { Name = Bob, Age = 43, Cash = 137 }

        /*
         * In Chapter 11, you learned about how you can use a delegate to reference
         * a method. In all of the examples of delegates that you've seen so far,
         * you assigned an existing method to a delegate.
         *
         * Anonymous methods are methods that you declare in a statement -- you
         * declare them using curly brackets { }, just like with anonymous types.
         *
         * You can learn more about anonymous methods here:
         * http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx
         */
        // Here's an anonymous method that writes an int and a string to the console.
        // Its declaration matches our MyIntAndString delegate (defined above), so
        // we can assign it to a variable of type MyIntAndString.
```

```
MyIntAndString printThem = delegate(int i, string s)
                              { Console.WriteLine("{0} - {1}", i, s); };
printThem(123, "four five six");
// Output: 123 - four five six

// Here's another anonymous method with the same signature (int, string).
// This one checks if the string contains the int.
MyIntAndString contains = delegate(int i, string s)
                             { Console.WriteLine(s.Contains(i.ToString())); };
contains(123, "four five six");
// Output: False

contains(123, "four 123 five six");
// Output: True

// You can dynamically invoke a method using Delegate.DynamicInvoke(),
// passing the parameters to the method as an array of objects.
Delegate d = contains;
d.DynamicInvoke(new object[] { 123, "four 123 five six" });
// Output: True

/*
 * A lambda expression is a special kind of anonymous method that uses
 * the => operator. It's called the lambda operator, but when you're
 * talking about lambda expressions you usually say "goes to" when
 * you read it. Here's a simple lambda expression:
 *
 *     (a, b) => { return a + b; }
 *
 * You could read that as "a and b goes to a plus b" -- it's an anonymous
 * method for adding two values. You can think of lambda expressions as
 * anonymous methods that take parameters and can return values.
 *
 * You can learn more about lambda expressions here:
 * http://msdn.microsoft.com/en-us/library/bb397687.aspx
 */

// Here's that lambda expression for adding two numbers. Its signature
// matches our CombineTwoInts delegate, so we can assign it to a delegate
// variable of type CombineTwoInts. Notice how CombineTwoInts's return
// type is int -- that means the lambda expression needs to return an int.
CombineTwoInts adder = (a, b) => { return a + b; };
Console.WriteLine(adder(3, 5));
// Output: 8

// Here's another lambda expression -- this one multiplies two numbers.
CombineTwoInts multiplier = (int a, int b) => { return a * b; };
Console.WriteLine(multiplier(3, 5));
// Output: 15

// You can do some seriously powerful stuff when you combine lambda
// expressions with LINQ. Here's a really simple example:
var greaterThan3 = new List<int> { 1, 2, 3, 4, 5, 6 }.Where(x => x > 3);
foreach (int i in greaterThan3) Console.Write("{0} ", i);
// Output: 4 5 6

Console.ReadKey();
    }
}
```

# #9. 使用DataContractSerializer串行化数据

把第9章介绍串行化的内容汇集在一起，我们的目标是让你对串行化如何工作的主要概念有一个很好的认识，另外了解BinaryFormatter类非常适合完成这个工作。不过这并不是串行化对象的唯一途径……这是有道理的，因为数据在对象中就可以有多种不同的表示方法。下面来看另外一种非常有用的串行化数据的方法：使用数据契约串行化器（Data Contract Serializer）。

数据契约串行化器（Data Contract Serializer）使用一个名为DataContractSerializer的类。这是Windows Communication Foundation (WCF)的一部分，这是Microsoft用于构建面向服务的应用的一个统一编程模型。下面给出一个例子，可以使用这个例子将Guy对象串行化为XML，以及由XML表示逆串行化为Guy对象。

```
/* Before you can serialize an object using the Data Contract Serializer, you need
 * to set up a data contract. The easiest way to do this is by marking the class with
 * the [Serializable] attribute. By default, the DataContractSerializer will write all
 * public read/write properties and fields. But what's really useful about the Data
 * Contract Serializer is that you can be a lot more specific about exactly what does
 * and doesn't get serialized. You can associate data with this particular class by
 * giving the contact a name and a namespace using named parameters.
 */
[DataContract(Name = "Guy", Namespace = "http://www.headfirstlabs.com")]
class SerializableGuy {
    // When you set up a specific data contract for a type -- like our
    // Guy class -- you mark each field or property that you want to
    // serialize with the [DataMember] attribute.
    [DataMember]
    public string Name { get; private set; }
    [DataMember]
    public int Age { get; private set; }
    [DataMember]
    public int Cash { get; private set; }

    // You can decide what members you want to serialize. We added two private int fields
    // called secretNumberOne and secretNumberTwo to our SerliazableGuy and initialized
    // them both to random numbers. secretNumberOne is marked with the [DataMember]
    // attribute, so it will be serialized as part of the data contract. But we didn't
    // mark secretNumberTwo, so it won't be. They're both returned as part of ToString().
    [DataMember]
    private int secretNumberOne = new Random().Next();
    // Since the secretNumberTwo field isn't marked with the [DataMember]
    // attribute, it's not part of the contract and won't be serialized.
    private int secretNumberTwo = new Random().Next();

    public SerializableGuy(string name, int age, int cash) {
        Name = name;
        Age = age;
        Cash = cash;
    }

    public override string ToString() {
        return String.Format("{0} is {1} years old and has {2} bucks [{3},{4}]",
            Name, Age, Cash, secretNumberOne, secretNumberTwo);
    }
}
```

可以用数据契约串行化器串行化任何[Serializable]类。不过如果使用[DataContract]和[DataMember]属性建立一个数据契约，会对能够串行化的对象有更多控制。

[DataContract]和[DataMember]属性在System.Runtime.Serialization命名空间中，所以Guy类最前面需要有一个“using System.RuntimeSerialization”指令。

```
using System;
using System.Text;
using System.Runtime.Serialization;
using System.IO;
using System.Xml;

class Program
{
    static void Main(string[] args)
    {
        /* Data contract serialization typically reads and writes XML data. You use a
         * DataContractSerializer object for data contract serialization. Its
         * WriteObject() method can write to a stream, or it can write to an object that
         * extends XmlDictionaryWriter, an abstract class that controls XML output and
         * can be extended to change the way the XML output is written. Objects are
         * deserialized using the ReadObject() method, which can read XML data from
         * a stream or an XmlDictionaryReader.
         */
        DataContractSerializer serializer = new DataContractSerializer(typeof(SerializableGuy));

        // We'll create a new SerializableGuy object and serialize it using a FileStream.
        SerializableGuy guyToWrite = new SerializableGuy("Joe", 37, 150);
        using (FileStream writer = new FileStream("serialized_guy.xml", FileMode.Create)) {
            serializer.WriteObject(writer, guyToWrite);
        }

        // We can open the file we just wrote and deserialize it into a new guy using ReadObject().
        // We'll use the XmlDictionaryReader.CreateTextReader() method to create an object that
        // reads XML data from a stream.
        SerializableGuy guyToRead = null;
        using (FileStream inputStream = new FileStream("serialized_guy.xml", FileMode.Open))
        using (XmlDictionaryReader reader =
            XmlDictionaryReader.CreateTextReader(inputStream, new XmlDictionaryReaderQuotas())) {
            guyToRead = serializer.ReadObject(reader, true) as SerializableGuy;
        }
        Console.WriteLine(guyToRead);
        // Output: Joe is 37 years old and has 150 bucks [1461194451,0]

        string xmlGuy = @"
<Guy xmlns=""http://www.headfirstlabs.com"" xmlns:i=""http://www.w3.org/2001/XMLSchema-instance"">
   <Age>43</Age>
   <Cash>225</Cash>
   <Name>Bob</Name>
   <secretNumberOne>54321</secretNumberOne>
</Guy>";
        byte[] buffer = UnicodeEncoding.UTF8.GetBytes(xmlGuy);

        using (XmlDictionaryReader reader =
            XmlDictionaryReader.CreateTextReader(buffer, new XmlDictionaryReaderQuotas()))
        {
            guyToRead = serializer.ReadObject(reader, true) as SerializableGuy;
        }
        Console.WriteLine(guyToRead);
        // Output: Bob is 43 years old and has 225 bucks [54321,0]
    }
}
```

试着在这里加一个断点，在监视窗口中增加reader和reader.Read()。多次刷新将reader.Read()时，会发生什么？

注意secretNumberOne字段逆串行化为契约的一部分，不过secretNumberTwo字段没有逆串行化。

打开serialized_guy.xml文件，你会看到类似这样的数据。这比读BinaryFormatter所写的内容要容易得多。不过也可以让DataContractSerializer写二进制数据。

这里取包含XML数据的串，将它转换为UTF-8编码的字节数组，再逆串行化为一个新的SerializableGuy对象。

可以在这里了解更多有关数据契约和数据契约串行化的内容：
http://msdn.microsoft.com/en-us/library/ms733127.aspx
另外可以在这里更多地了解Windows Communication Foundation：
http://msdn.microsoft.com/en-us/library/dd456779.aspx

# #10. LINQ to XML

XML或可扩展标记语言（Extensible Markup Language）是一种将复杂数据表示为文本的文件和数据流格式。.NET框架为创建、加载和保存XML文件提供了一些相当强大的工具。一旦得到XML数据，就可以使用LINQ进行查询。在文件最前面增加"using System.Xml.Linq;"，输入以下方法，它会生成一个XML文档，存储Starbuzz顾客的个人数据。

```
private static XDocument GetStarbuzzData() {
    XDocument doc = new XDocument(
       new XDeclaration("1.0", "utf-8", "yes"),
       new XComment("Starbuzz Customer Loyalty Data"),
       new XElement("starbuzzData",
           new XAttribute("storeName", "Park Slope"),
           new XAttribute("location", "Brooklyn, NY"),
           new XElement("person",
               new XElement("personalInfo",
                   new XElement("name", "Janet Venutian"),
                   new XElement("zip", 11215)),
               new XElement("favoriteDrink", "Choco Macchiato"),
               new XElement("moneySpent", 255),
               new XElement("visits", 50)),
           new XElement("person",
               new XElement("personalInfo",
                   new XElement("name", "Liz Nelson"),
                   new XElement("zip", 11238)),
               new XElement("favoriteDrink", "Double Cappuccino"),
               new XElement("moneySpent", 150),
               new XElement("visits", 35)),
           new XElement("person",
               new XElement("personalInfo",
                   new XElement("name", "Matt Franks"),
                   new XElement("zip", 11217)),
               new XElement("favoriteDrink", "Zesty Lemon Chai"),
               new XElement("moneySpent", 75),
               new XElement("visits", 15)),
           new XElement("person",
               new XElement("personalInfo",
                   new XElement("name", "Joe Ng"),
                   new XElement("zip", 11217)),
               new XElement("favoriteDrink", "Banana Split in a Cup"),
               new XElement("moneySpent", 60),
               new XElement("visits", 10)),
           new XElement("person",
               new XElement("personalInfo",
                   new XElement("name", "Sarah Kalter"),
                   new XElement("zip", 11215)),
               new XElement("favoriteDrink", "Boring Coffee"),
               new XElement("moneySpent", 110),
               new XElement("visits", 15))));
    return doc;
}
```

可以使用XDocument创建一个XML文件，这包含可以使用DataContractSerializer读写的XML文件。

XMLDocument对象表示一个XML文档。这是System.Xml.Linq命名空间的一部分。

使用XElement对象来创建XML树下的元素。

Microsoft关于LINQ和LINQ to XML提供了大量在线文档。可以在以下地址更多地了解LINQ to XML和System.Xml.Linq命名空间中的类：
http://msdn.microsoft.com/en-us/library/bb387098.aspx

## 保存和加载XML文件

可以向控制台写XDocument对象，或者把它保存到一个文件中，另外可以将一个XML文件加载到XDocument对象：

```
XDocument doc = GetStarbuzzData();
Console.WriteLine(doc.ToString());
doc.Save("starbuzzData.xml");
XDocument anotherDoc = XDocument.Load("starbuzzData.xml");
```

XDocument对象的Load()和Save()方法读写XML文件。ToString()方法将其中的所有内容表示为一个大的XML文档。

## 查询数据

下面是一个简单的LINQ查询，使用XDocument查询Starbuzz数据：

```
var data = from item in doc.Descendants("person")
    select new { drink = item.Element("favoriteDrink").Value,
        moneySpent = item.Element("moneySpent").Value,
        zipCode = item.Element("personalInfo").Element("zip").Value };
foreach (var p in data)
    Console.WriteLine(p.ToString());
```

Descendants()方法返回一个对象的引用，可以把它直接插入到LINQ查询。

你已经知道LINQ允许调用方法，而且可以使用方法作为查询的一部分，这一点也适用于Element()方法。

还可以完成更复杂的查询：

```
var zipcodeGroups = from item in doc.Descendants("person")
    group item.Element("favoriteDrink").Value
    by item.Element("personalInfo").Element("zip").Value
        into zipcodeGroup
        select zipcodeGroup;
foreach (var group in zipcodeGroups)
    Console.WriteLine("{0} favorite drinks in {1}",
                group.Distinct().Count(), group.Key);
```

Element()返回一个XElement对象，可以使用这个对象的属性来检查XML文档的某些特定值。

## 从RSS提要读取数据

用LINQ to XML可以实现一些很强大的功能。下面是一个简单的查询，要从我们的博客读取文章：

```
XDocument ourBlog = XDocument.Load("http://www.stellman-greene.com/feed");
Console.WriteLine(ourBlog.Element("rss").Element("channel").Element("title").Value);
var posts = from post in ourBlog.Descendants("item")
    select new { Title = post.Element("title").Value,
        Date = post.Element("pubDate").Value};
foreach (var post in posts)
    Console.WriteLine(post.ToString());
```

XDocument.Load()方法有很多重载方法。这个重载版本从一个URL取得XML数据。

创建一个新的控制台应用，确保在最前面增加一行"using System.Xml.Linq;"，把这个查询键入到事件处理程序中，看看会向控制台写出什么输出。

这里使用了我们的博客（Building Better Software）的URL：

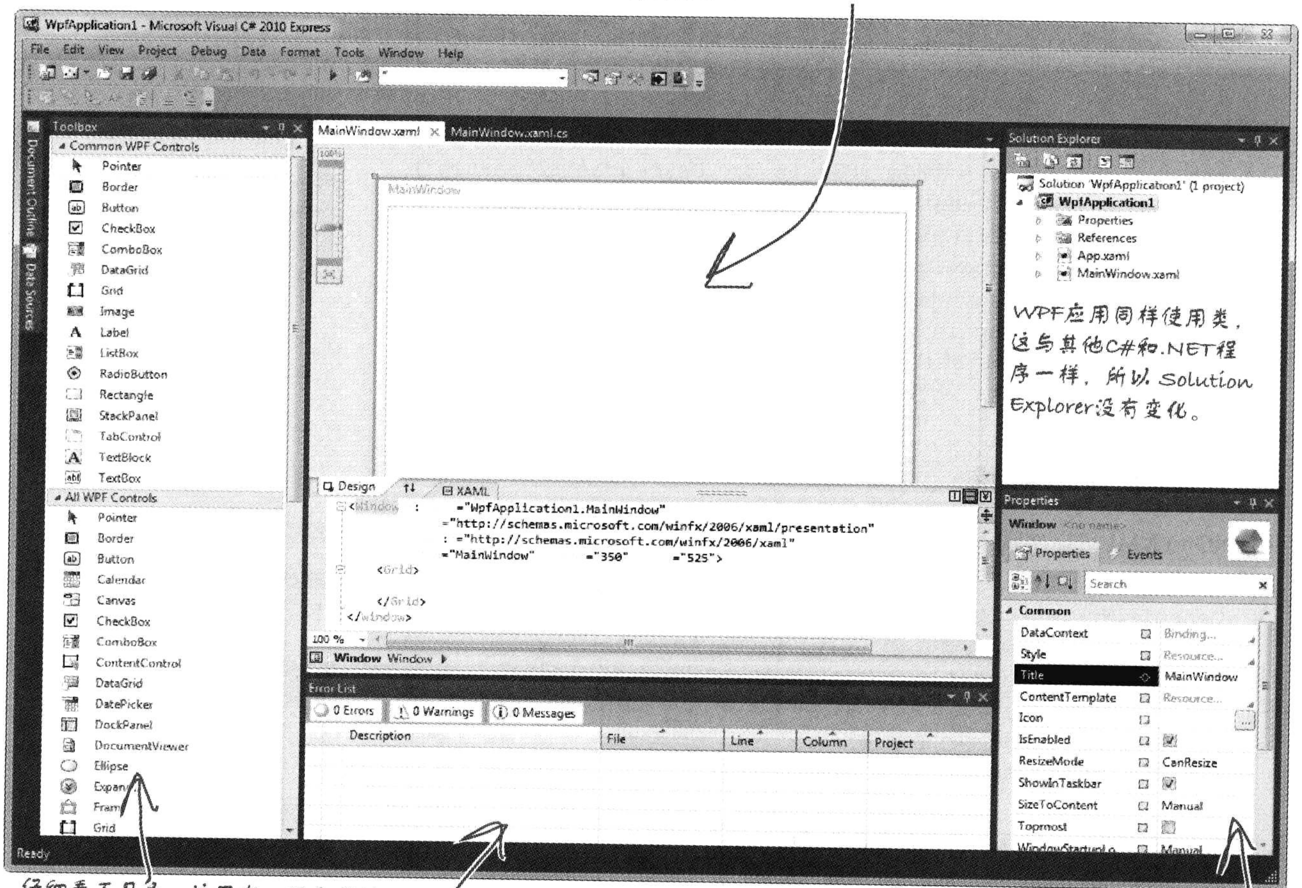
http://www.stellman-greene.com/

# #11. Windows表现基础库

Windows表现基础库（Windows Presentation Foundation，WPF）是Microsoft为构建可视化应用提供的下一代平台。它确实是无与伦比，提供了基于XML的布局、可伸缩的控件、一个全新的控件系统、2D/3D图形化和动画、文本流和文档格式化，甚至还有一个使用了WPF的跨平台web浏览器插件。

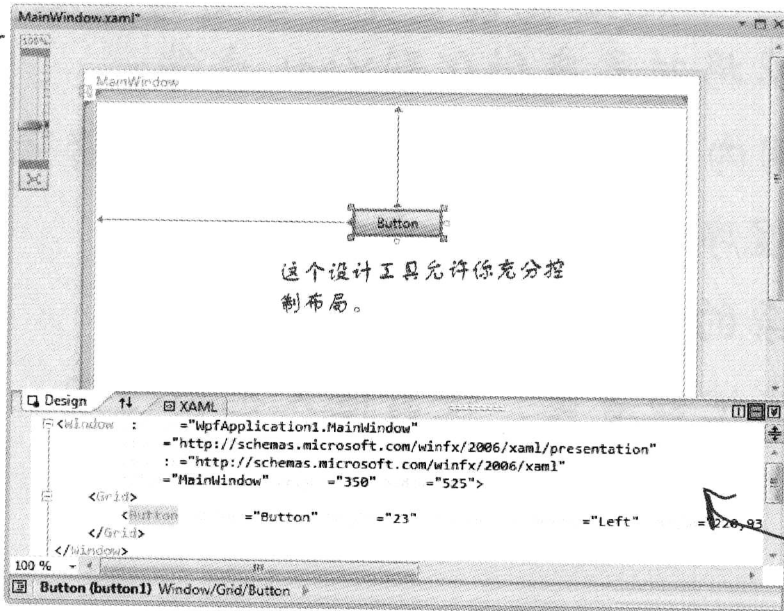
遗憾的是，尽管WPF确实很酷，而且技术含量很高，但对于C#教学来说却不是一个很好的工具。我们的目标是：让C#概念尽可能快、尽可能容易地进入你的大脑。

可以用一点时间创建一个新的WPF应用。只需使用IDE创建一个新工程，不过不是创建一个新的Windows Forms Application工程。相反，选择WPF Application。可以立即注意到IDE中的一个差别：



从工具条将一个按钮拖到窗体上。如果这是一个Windows Forms应用，IDE就会在Form1.Designer.cs中增加代码向Form1对象增加一个控件。不过WPF不同，它使用一种基于XML的语言XAML来定义用户界面如何布局、如何与对象交互等。

上下拖动这个滚动条来放大和缩小用户界面。放大到非常大时，用户界面看上去仍然很好，不会失常。




XAML代表“可扩展应用标记语言”（Extensible Application Markup Language），这是一种基于XML的语言，WPF使用XAML确定所有控件以及其他UI元素放在哪里。

IDE有一个非常强大的XML编辑器，为处理XAML专门做了优化。

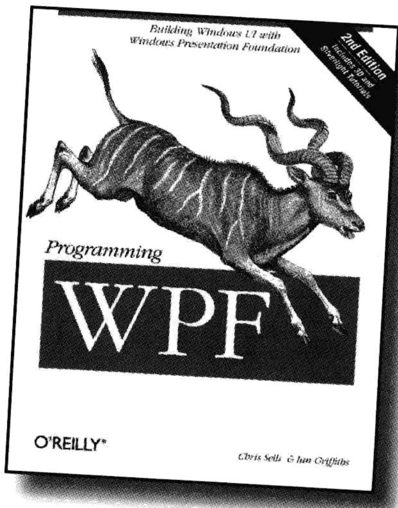
进入XAML编辑器，在XAML编辑器中键入下面加粗的一行来增加第二个按钮。你会注意到IDE的智能提示窗口非常强大，可以帮助你输入所有XML标记。

```
<Grid>
    <Button Height="23" Margin="98,43,105,0" Name="button1"
                   VerticalAlignment="Top" Click="button1_Click">Button</Button>
    <Button Height="23" Margin="5,5,100,20" Name="button2"
                   VerticalAlignment="Top" Click="button2_Click">Another button</Button>
</Grid>
```

在这一行的“Click=”button2_Click””部分，不要键入事件处理程序的名字，而使用弹出的智能提示窗口告诉IDE增加一个新的事件处理程序。完成这一行后，你会看到设计工具中出现一个新的按钮。切换到Window1.xaml.cs页，可以看到有一个新的button2_Click方法。

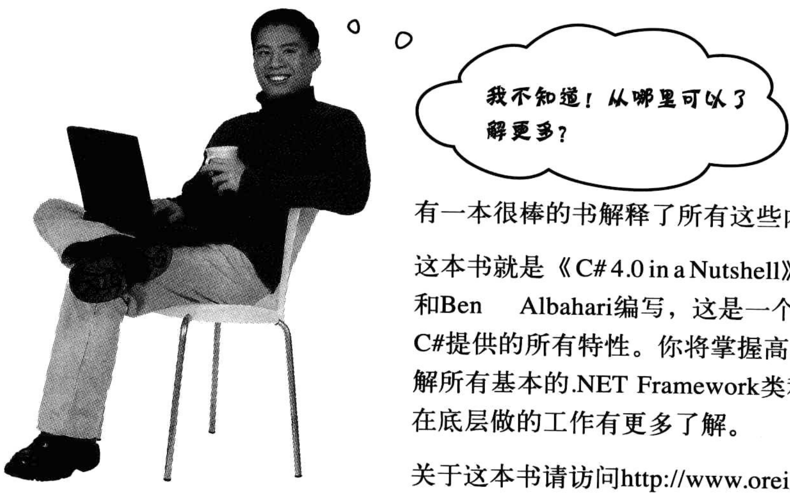
关于WPF和XAML，我们在这里只谈这么多。不过，说到学习WPF的工具，强烈建议你看看Chris Sells和Ian Griffiths写的《Programming WPF》。可以从O’Reilly网站了解这本书（http://www.oreilly.com/）。

可以通过一个项目来尝试一下WPF：试着用WPF构建第1章的Contact应用。大多数步骤都完全相同！只是要确保使用一个Image而不是PictureBox，另外要使用属性窗口中的“事件”页增加一个MouseDown事件处理程序，因为它没有Click事件。另外，不要使用“mbox”代码段，应当使用MessageBox.Show()。



## 你知道C#和.NET Framework能够……

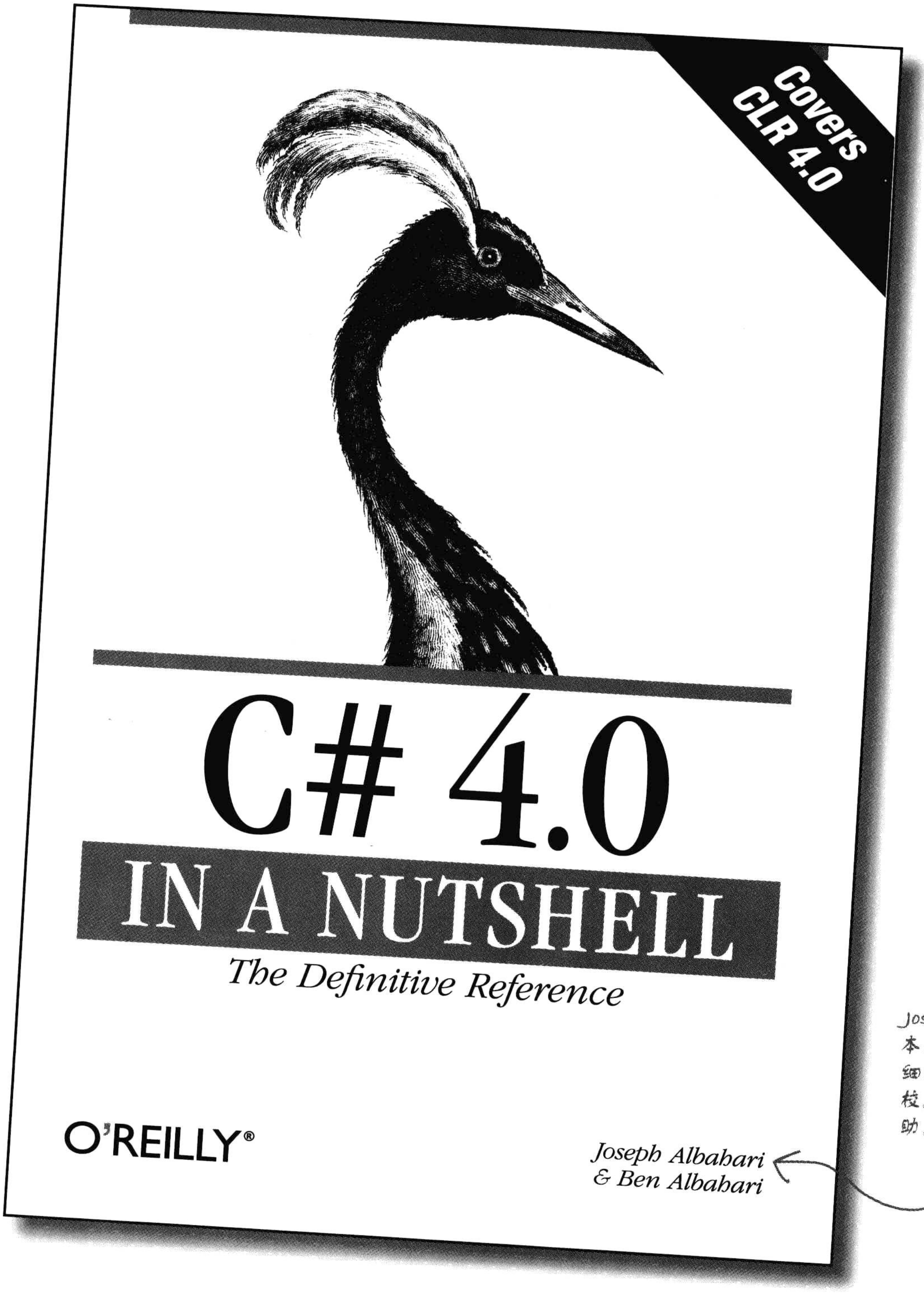
- 利用高级LINQ查询能更充分地处理数据？而且可以将对象串行化到XML文件？
- 可以使用内置类访问网站和其他网络资源？
- 允许为程序增加高级加密和安全性？
- 创建复杂的多线程应用？
- 可以适当地部署类，使其他人也能使用？
- 使用正则表达式完成高级文本搜索？




有一本很棒的书解释了所有这些内容！

这本书就是《C# 4.0 in a Nutshell》，由Joseph Albahari和Ben Albahari编写，这是一个全面的指南，介绍了C#提供的所有特性。你将掌握高级C#语言特性，会了解所有基本的.NET Framework类和工具，而且将对C#在底层做的工作有更多了解。

关于这本书请访问http://www.oreilly.com/。




Joseph Albahari对这本书的第一版做了细致全面的技术审校。感谢你的大力帮助，Joe！

# 索引

## 符号


() (parentheses) （() (括号)），184

* (asterisk) （* (星号)） 46

@ (at sign) (@ (at符号)) 411～412, 423

// (double-slash comments)（// (双斜线注释)） 66

" " (empty string) （ “ “ (空串)） 62, 74

! (exclamation point) （! (感叹号)） 475

? (question mark)（? (问号)） 673

‘ (single quotes) （ ‘ (单引号)） 127

/// (triple-slash comments)（/// （三斜线注释)） 736～737

, (comma)（, (逗号)） 329, 344

?: conditional operator （?:条件操作符） 739

: operator （:操作符） 234, 274, 330

?? null coalescing operator （?? null结合操作符） 739

! operator （!操作符） 62, 260

!= operator （!=操作符） 68

& operator （&操作符） 740

&& operator （&&操作符） 68, 77, 739～740

* operator （*操作符） 62, 138

*= operator （*=操作符） 62, 101, 138

+ operator （+操作符） 62, 131, 353

++ operator （++操作符） 62, 739

+= operator （+=操作符）

  = operator and （=操作符） 534

  event handling （事件处理） 513, 515, 517, 535

  functionality （功能） 62

- operator （-操作符） 132, 138

-- operator （--操作符） 62

-= operator （-=操作符） 138

/ operator （/操作符） 62

= operator （=操作符）

  += operator and （+=操作符） 534

  == operator and （==操作符） 67

  casting support （强制转换支持） 132

  return values and （返回值） 739

== operator （==操作符）

  = operator and （=操作符） 67

  conditional testing （条件测试） 68, 70

  examples （示例） 77, 750～752

^ operator （^操作符） 740

| operator （|操作符） 740

|| operator （||操作符） 68, 739

; (semicolon) (参见semicolon (;))

< operator （<操作符） 68, 70

<< operator （<<操作符） 741

<> (angle brackets) （<> (尖括号)） 335, 363

> operator （>操作符） 68, 70

>> operator （>>操作符） 741

\\ (double backslash)（\\ (双反斜线)） 423

\n (line break) (参见line break (\n))

\r (return character)（\r (回车符)） 373, 423

\t (tab)（\t (制表符)） 127, 411, 423

{} (curly brackets)（{} (大括号)） 51, 56, 65～66, 73

~ (tilde)（~ (波浪号)） 654

~ operator （~操作符） 740

# A

abstract classes（抽象类）
building a house application（构建一个房屋应用） 308～322
defined（定义） 296～297
examples （示例） 298
generic collections（泛型集合） 335
Stream class（Stream类） 409
abstract methods（抽象方法） 296, 299
abstraction（抽象） 306
access modifiers（访问修饰符）
changing visibility（改变可见性） 292～293
on class declarations（类声明） 744
defined（定义） 291
adapters（适配器） 31
Add >> Class feature（Add >> Class 特性） 331
Add New Item window（Add New Item窗口） 18
address book application（地址簿应用） 4～5, 9 ～16, 20～34
adventure game application（冒险游戏应用） 385～406
Albahari, Ben, 766～767
Albahari, Joe, 711, 748, 766
allocated resources（分配的资源） 427
angle brackets <>（尖括号<>） 335, 363
animated beehive simulator（动画蜂巢模拟系统）
adding forms（增加表单） 570
adding timers（增加定时器） 574～576
architecture（体系结构） 544, 557
behavior considerations（行为考虑） 568～569
building（构建） 545～560, 562～567
callback technique（回调技术） 578
collections（集合） 581～582
controls and（控件） 592～593, 596, 599～607
delegates and（委托） 578～579
LINQ support（LINQ支持） 582～583
List<T> class（List<T>类） 581～582
opening/saving（打开/保存） 585～587
overhauling（全面检查） 635～639
overview（复习） 543, 554
performance considerations（性能考虑） 615～617
Renderer class and（Renderer类） 594～595, 607, 609～611
testing（测试） 577～579, 584, 614
as turn-based system（基于回合的系统） 561
working with groups（处理组） 580
anonymous methods（匿名方法） 758～759
anonymous types（匿名类型） 703, 709, 758～759
application development（应用开发）
adding loops（增加循环） 65, 69
adding statements（增加参数） 66
adding to auto-generated code（增加到自动生成的代码） 2, 11, 15
auto-generated code and（自动生成的代码）, 2, 11, 15, 73
debugging code（调试代码） 16, 469
deployment and（部署） 35
designing for intuitive use（设计支持外观） 32～33
developing user interface（开发用户界面） 12～13
embedding databases（嵌入数据库） 18
modifying generated code（修改生成的代码） 11
overview（概述） 6～7, 44～45
source code files（源代码文件） 44
testing programs（测试程序） 34
tools supporting（工具支持） 46～47
turn-based systems（基于回合的系统） 386
using classes（使用类） 89～91
using variables（使用变量） 60～61
Application.DoEvents() method（Application.DoEvents()方法） 746
args parameter（args参数） 457
ArgumentException, 492, 495, 565
ArgumentOutOfRangeException, 501
arguments（参数）
command-line（命令行） 456～457
defined（定义） 133, 669
event（事件） 512
named（命名） 672
type（类型） 340
Array.Reverse() method（Array.Reverse()方法） 448
arrays（数组）
creating deck of cards（创建一副牌） 333～334
defined（定义） 150
determining length（确定长度） 151
exception handling（异常处理） 470
foreach loops（foreach循环） 339
lists and（列表） 336, 338, 343
of object references（对象引用） 151
reference variables and（引用变量） 151
static method for（静态方法） 448
as keyword（as关键字）
coffeemaker example（咖啡机示例） 286

functionality（功能）283
value types and（值类型）668, 677
ASCII characters（ASCII字符）457～458
assemblies（程序集）291, 742～743
asterisk (*)（星号（*)) 46
at sign (@)（at符号（@)) 411～412, 423
attributes（属性）443
audio files（音频文件）14
auto-generated code（自动生成的代码）
adding to（增加到）15
overview（概述）2, 11
undoing（撤销）73
auto-generating record IDs（自动生成记录ID）21
automatic properties（自动属性）
adding（增加）205
beehive simulator example（蜂巢模拟系统示例）546, 550～551
event arguments and（事件参数）520
functionality（功能）280, 549


# B


BackgroundWorker component（BackgroundWorker组件）746～748
backing fields（后备字段）
accessor methods and（存取方法）209
beehive simulator example（蜂巢模拟系统示例）550
defined（定义）203
examples（示例）206
resizing（调整大小）219
this keyword（this关键字）and, 208
base classes（基类）
choosing（选择）229
constructors and（构造函数）251
extending（扩展）233, 251
generic collections（泛型集合）335
inheritance and（继承）226, 229～230, 234, 237, 239, 250～251
for objects（对象的）, 353
subclasses and（子类）233, 239, 250
upcasting and（向上强制转换）285
virtual keyword（virtual关键字）, 238
base keyword（base关键字）, 250
baseball simulator application（棒球模拟系统示例）
building（构建）508～514, 518～521
creating event handlers（创建事件处理程序）516～517
beehive management system（蜂巢管理系统）
building（构建）257～267
data storage（数据存储）328～329
exception handling（异常处理）465～467, 491～492
inheritance（继承）270～275
interfaces and（接口）279～283
(参见 animated beehive simulator)
beehive simulator (参见animated beehive simulator)
behavior, classes and（行为，类）17, 568～569
bin folder（bin文件夹）34
binary format（二进制格式）
serializing objects in（串行化对象）446
working with（处理）453
writing in（写入）449
BinaryFormatter object（BinaryFormatter对象）
Deserialize() method（Deserialize()方法）442, 445, 478, 483
exception handling（异常处理）477～478
[Serializable] attribute（[Serializable]属性）445
Serialize() method（Serialize()方法）442, 445
BinaryReader class（BinaryReader类）
functionality（功能）450
ReadBytes() method（ReadBytes()方法）450～451
ReadChar() method（ReadChar()方法）450
ReadInt32() method（ReadInt32()方法）450
ReadSingle() method（ReadSingle()方法）450
ReadString() method（ReadString()方法）450
BinaryWriter class（BinaryWriter类）449
binding navigator（绑定导航工具）31
bit data type（bit数据类型）25
Bitmap class（Bitmap类）
double buffering（双缓冲）634
overloaded constructors（重载构造函数）619
resizing bitmaps（调整位图大小）618
storage considerations（存储考虑）617, 619
black boxes, objects as（黑盒，对象）199～200
bool data type（bool数据类型）
defined（定义）61～62, 126
memory considerations（内存考虑）128
break statement（break语句）436, 738
breakpoints（断点）

adding（增加） 64, 347, 474
beehive simulator example（蜂巢模拟系统示例） 603
determining placement（确定位置） 476
Brush object（Brush对象） 622, 632
Build menu（Build菜单） 35
building programs (参见program builds)
built-in generic collections（内置泛型集合） 377
built-in objects（内置对象） 419
buttons（按钮）
adding to forms（增加到窗体） 114, 357, 598, 602
BackColor property（BackColor属性） 49
Click event（Click事件） 513
DisplayStyle property（DisplayStyle属性） 570
Enabled property（Enabled属性） 367～368
event handlers and（事件处理程序） 524～525, 585
interacting with objects（与对象交互） 115
Location property（Location属性） 49
Name property（Name属性） 49, 367
OnClick() event（OnClick()事件） 535
Size property（Size属性） 49
Text property（Text属性） 49
by keyword（by关键字）, 708
byte data type（byte数据类型）
converting strings to（字符串转换为） 423
converting to strings（转换为字符串） 457
defined（定义） 126
moving data around（移动数据） 448
reading from streams（从流读取） 456～457
streams and（流） 410
byte order mark（字节序标记） 458


# C


C#
benefits using（使用的好处） 2～3
reserved words（保留字） 156, 164
call stacks（调用栈） 477, 498
callback methods（回调方法） 532～534
callback technique（回调技术）
beehive simulator example（蜂巢模拟系统示例） 578
debugging（调试） 535
defined（定义） 532
delegates and（委托） 532～536
events and（事件） 536
Golden Crustacean application（黄金蟹应用） 538～539
camelCase, 211
Captain Amazing application（Captain Amazing应用） 647～653, 677, 691～692
cards, deck of (参见deck of cards application)
case statement（case语句） 435～437
casting（强制转换）
automatically（自动地） 131～132
defined（定义） 130
examples （示例） 330, 373
catch block (参见try/catch blocks)
chaining（串链）
event handlers（事件处理程序） 535
events（事件） 524～525
objects（对象） 515
streams（流） 416
char data type（char数据类型）
converting to strings（转换为字符串） 455
defined（定义） 127
memory considerations（内存考虑） 128
StreamReader class（StreamReader类） 455
Unicode standard（Unicode标准） 447
Character Map（字符映射表） 446～447, 450
CheckBox control（CheckBox控件）
event planning example（事件规划示例） 221
name considerations（命名考虑） 185
Text property（Text属性） 32
child windows, defined（子窗口，定义） 592
class diagrams（类图）
adding fields（增加字段） 100
building（构建） 106, 108
constants and（常量） 546
defined（定义） 91
grouping classes（类分组） 231
interfaces on（接口） 281
methods and（方法） 91
moving down in（下移） 239, 243
parameters on（参数） 394
return values on（返回值） 394
class hierarchy（类层次体系）
beehive management system（蜂巢管理系统） 271
creating（创建） 232
defined（定义） 227, 233

overriding methods and（覆盖方法） 249
upcasting（向上强制转换） 285
class members（类成员） 291～292
class skeleton（类骨架） 546
classes（类）
abstract（抽象） 296～297
adding（增加） 54
adding components（增加组件） 605
adding constants（增加常量） 546
allocated resources（分配的资源） 427
automatic properties（自动属性） 205
building overloaded methods（建立重载方法） 357
building programs with（用来构建程序） 89～91
collection initializers and（集合初始化方法）, 365
concrete（具体） 296
constructors and（构造函数） 207, 209
controlling access（控制访问） 192～193
controls and（控件） 590
creating instances（创建实例） 94, 103, 217
creating objects（创建对象） 93～94
declaring（声明） 52
defined（定义） 50～51
defining events（定义事件） 512
delegates and（委托） 527
design ideas（设计想法） 118
determining behavior（确定行为） 17, 568～569
empty（空） 512
encapsulating（封装） 199～201, 459, 549
enum data type（enum数据类型） and, 331
examples （示例） 53, 112
exception handling and（异常处理） 491
extension methods and（扩展方法） 678～679
generic collections（泛型集合） 335, 340
grouping（成组） 231
hiding information（隐藏信息） 197～200
implementing interfaces（实现接口） 280～282, 288
inheritance and（继承） 225～227, 304, 590, 679
instantiating（实例化） 295, 298
interface requirements（接口需求） 272, 275
lists and（列表） 343
members（成员） 291
methods and（方法） 50, 89
name considerations（命名考虑） 104～105, 118
namespace considerations（命名空间考虑） 59
natural structure（自然结构） 106
organizing（组织） 108
partial（部分） 73
planning（规划） 106
polymorphism and（多态） 307
private fields and（私有字段） 191, 193
[Serializable] attribute（[Serializable]属性） 443～444
serializing（串行化） 443
sharing methods（共享方法） 90
statements and（语句） 73
structs and（struct） 676
tracking statistics（跟踪统计） 162
using statements and（using语句） 73, 496
XML comments（XML注释） 736～737
(参见 collections)
clauses (query)（子句（查询）） 690
(参见 specific clauses)
CLR (Common Language Runtime)（通用语言运行时）
defined（定义） 45
finalizers and（最终化方法） 654, 656
functionality （功能） 667
garbage collection and（垃圾回收） 155
just-in-time compiler（即时编译器） 745
memory and（内存） 667
code, writing (参见application development)
code blocks（代码块）
curly brackets and（大括号） 56, 73, 219
defined（定义） 56, 73
methods as（方法作为） 15
coffeemaker application（咖啡机应用） 284～287
collection initializers（集合初始化方法）
defined（定义） 344
examples （示例） 345
IEnumerable<T> interface（IEnumerable<T>接口） 355
object initializers and（对象初始化方法） 344
populating classes（填充类） 365
collections（集合）
adding keys to（增加键） 365
beehive simulator example（蜂巢模拟系统示例） 581–582
combining（合并） 356, 703～704
CurrentSize property（CurrentSize属性） 583
as databases（作为数据库0 581～582
defined（定义） 335
foreach loops（foreach循环） 339
generic（泛型） 335, 340, 343, 377
IEnumerable<T> interface and（IEnumerable<T> 接口） 689, 694
LINQ support（LINQ支持） 582～583, 689, 705
performing calculations on（完成计算） 694

storing（存储） 335
collision detection（碰撞检测） 727
colon (:) operator （冒号(:)操作符） 234, 274, 330
Color class（Color类）
FromArgb() method（FromArgb()()方法） 76
Transparent property（Transparent属性） 615
columns（列）
adding to tables（增加到表） 20, 22～24
defined（定义） 20
forcing values（强制值） 25
ComboBox control（ComboBox控件）
DropDownStyle property（DropDownStyle属性） 259
SelectedIndexChanged event（SelectedIndexChanged事件） 365
SelectedItem property（SelectedItem属性） 365
comma (,)（逗号（,）） 329, 344
command-line arguments（命令行参数） 456～457
comments（注释）
//, 66
///, 736～737
adding（增加） 66, 739
debugger and（调试工具） 63
exception handling and（异常处理） 498
functionality （功能） 63
XML, 736～737
Common Intermediate Language（通用中间语言） 745
Common Language Runtime (参见CLR (Common Language Runtime))
comparison operators（比较操作符） 70
compilation (参见program compilation)
components（组件）
adding to classes（增加到类） 605
defined（定义） 420, 605
compound operators（复合操作符） 138
concatenating strings（连接字符串） 132, 353
concrete classes（具体类） 296
concrete methods（具体方法） 296, 299
conditional expressions（条件表达式） 757
conditional operators（条件操作符） 68, 739
conditional tests（条件测试）
comparison operators and（比较操作符） 70
defined（定义） 68, 73
examples （示例） 69
if/else statements（if/else语句） 67～68
Console applications（控制台应用）
command-line arguments（命令行参数） 456
creating（创建） 244～245
delegate types and（委托类型） 527
forms and（窗体） 244
Main() method（Main()方法） 244～245
ReadKey() method（ReadKey()方法） 245, 345, 347, 351～352
WriteLine() method（WriteLine()方法） 204, 339, 351, 373, 447
constants（常量）
adding to classes（增加到类） 546
class diagrams and（类图） 546
defined（定义） 182
enum data type（enum数据类型） and, 343
constructors（构造函数）
adding to classes（增加到类） 207, 209
base classes and（基类） 251
building（构建） 437, 547
converting arrays to lists（数组转换为列表） 343
defined（定义） 207, 483
exception handling（异常处理） 483
initializing private fields（初始化私有字段） 207
parameters for（参数） 207～208
return types and（返回类型） 208
return values and （返回值） 207, 209
streams and（流） 415
subclasses and（子类） 251
(参见 overloaded constructors)
Containers toolbox（Containers工具箱） 425
continue keyword（continue关键字）, 738
controls（控件）
accessing properties（访问属性） 13
adding to forms（增加到窗体） 12, 30～31, 590, 607
adding to toolbox（增加到工具箱） 604
beehive simulator example（蜂巢模拟系统示例） 592～593, 599～607
binding to tables（绑定到表） 31
BringToFront() method（BringToFront()方法） 607
building animated（构建动画控件） 599～607
classes and（类） 590
CreateGraphics() method（CreateGraphics()方法） 618
creating（创建） 590
database-driven（数据库驱动） 30～31
dialog boxes as（对话框） 420

displaying in toolbox（工具箱中显示） 160, 599
Dispose() method（Dispose()方法） 603, 609～610
editing pre-existing（编辑既有控件） 613
event handlers（事件处理程序） 221, 535
functionality （功能） 591
inheritance and（继承） 600～601
initializing（初始化） 208
Invalidate() method（Invalidate()方法） 629
Maximum property（Maximum属性） 629
Minimum property（Minimum属性） 629
Name property（Name属性） 358
non-visual（不可见） 161, 420
as objects（对象） 591
Orientation property（Orientation属性） 629
Paint event（Paint事件） 628
redrawing（重绘） 628
Remove() method（Remove()方法） 609
removing from forms（从窗体删除） 590
repainting（重绘） 631
ResizeCells() method（ResizeCells()方法） 617
ResizeImage() method（ResizeImage()方法） 617
Size property（Size属性） 611
TickStyle property（TickStyle属性） 629
Value property（Value属性） 127, 629
visual display suitability（可视化显示适应性） 596
(参见 specific controls)

Controls collection（Controls集合） 591, 598

converting（转换）
arrays to lists（数组转换为列表） 343
byte arrays to strings（字节数组转换为字符串） 457
char to strings（字符转换为字符串）, 455
data types automatically（自动转换数据类型） 131, 156
hex to decimal（十六进制转换为十进制） 446, 450, 473
lists to stacks（列表转换为栈） 380
objects to strings（对象转换为字符串） 353～354
queues to lists（队列转换为列表） 380
stacks to queues（栈转换为队列） 380
strings to byte arrays（字符串转换为字节数组） 423
variables to strings（变量转换为字符串） 132, 185

ConvertsIntToString delegate type（ConvertsIntToString委托类型） 527

Convert.ToDateTime() method（Convert.ToDateTime()方法） 431

covariance, defined（协变，定义） 356

CryptoStream class（CryptoStream类） 416

Ctrl-F5 key combination（Ctrl-F5按键组合） 352

Ctrl-Tab key combination（Ctrl-Tab按键组合） 46

curly brackets {}（大括号{}）
automatically indenting（自动缩进） 113
for code blocks（代码块） 56, 73, 219
collection initializers and（集合初始化方法） 344
enumerator lists（枚举项列表） 329
matching pairs（匹配对） 51, 65
methods and（方法） 66


# D


Data menu（Data菜单）
Add New Data Source... option（Add New Data Source...选项） 28
Show Data Sources option（Show Data Sources选项） 30

Data Source Configuration Wizard（数据源配置向导） 18

data sources（数据源）
adding database-driven controls（增加数据库驱动控件） 30
configuring（配置） 29
connecting forms to databases（将窗体连接到数据库） 28～29
defined（定义） 28
showing（显示） 30

data storage（数据存储）
about（关于） 7
collections and（集合） 335
enums and lists（enum和列表） 335, 343
generic collections and（泛型集合） 340
SQL databases（SQL数据库） 18～19, 34
storing keys and values（存储键和值） 363
structs and（struct） 663
Unicode standard（Unicode标准） 447

data types（数据类型）
casting values（强制转换值） 130～132
converting automatically（自动转换） 131, 156
declaring variables（声明变量） 61, 66
defined（定义） 20
Dictionary class（Dictionary类） 363
generic collections and（泛型集合） 335, 340
key-value pairs and（键-值对） 364
lists and（列表） 343
method parameters/arguments（方法行参/实参） 133
overview（概述） 126～127

Database Explorer, 18

databases（数据库）
binding controls to（绑定控件） 30～31
collections as（集合） 581～582
connecting forms to（连接窗体） 17, 28～29
creating（创建） 3, 18～19
defined（定义） 7
defining fields（定义字段） 22
embedding into programs（嵌入在程序中） 18
inserting data into（插入数据） 26～27
LINQ support（LINQ支持） 582～583, 705～708
overwriting data（重写数据） 34
saving records（保存记录） 27
SQL databases（SQL数据库） 18～19, 34
viewing in Solution Explorer（在Solution Explorer中查看） 18
(参见 tables)
DataContractSerializer class（DataContractSerializer类） 760～761
DateTime class（DateTime类）
functionality （功能） 574
Now property（Now属性） 574
TryParse() method（TryParse()方法） 674
dead objects（死对象） 655
Debug menu（Debug菜单）
Continue option（Continue选项） 64
Start Debugging option（Start Debugging 选项） 45, 47, 64
Step Over option（Step Over选项） 64
Stop Debugging option（Stop Debugging选项） 77
Windows menu（Windows菜单） 204
Debug toolbar（Debug工具条）
Break All button（Break All按钮） 473, 476
Continue button（Continue按钮） 473, 483
expanding（展开） 473
Hex button（Hex按钮） 473
Locals button（Locals按钮） 473
Next Statement button（Next Statement按钮） 473, 482–483, 485
Restart button（Restart按钮） 473
Step Into button（Step Into按钮） 473～474, 482, 529
Step Out button（Step Out按钮） 473
Step Over button（Step Over按钮） 473～474, 482
Stop button（Stop按钮） 473
debugging programs/debugger（调试程序/调试工具）
adding breakpoints（增加断点） 64, 347, 474
defined（定义） 16, 45
Error List window（Error List窗口） 47
exploring callbacks（分析回调） 535
exploring delegates（分析委托） 529, 535
exploring events（分析事件） 535
finding exceptions（查找异常） 469, 481
hovering over fields（停在字段上） 476
hovering over variables（停在变量上） 64
passing command-line arguments（传递命令行参数） 456
starting process（开始进程） 64
stepping through code（单步跟踪代码） 64, 339
testing programs（测试程序） 34
viewing variable value changes（查看变量值改变） 63～64
(参见 exception handling; Watch window)
decimal data type（decimal数据类型）
converting from hex（从十六进制转换）, 446, 450, 473
defined（定义） 127
memory considerations（内存考虑） 128
deck of cards application（扑克牌应用）
creating（创建） 331～335
practice exercise（实践练习） 358～362
serializing/deserializing（串行化/逆串行化） 444～445
declaration（声明）
access modifiers on（访问修饰符） 744
allocated resources and（分配的资源） 427
for classes（类） 52
defined（定义） 52
for dictionaries（字典） 363
for List<T> class（List<T>类） 340
for methods（方法） 52
for private fields（私有字段） 191
for reference variables（引用变量） 150
using statements and（using语句） 428
for variables（变量） 60, 66, 76, 116, 150
decoding data（解码数据） 423
default statement（default语句） 436
delegate types（委托类型） 527, 535
delegates（委托）
adding（增加） 528～529, 535
beehive simulator example（蜂巢模拟系统示例） 578～579
callback technique（回调技术） 532～536
checking for null value（检查null值） 535
creating（创建） 527
debugging（调试） 529, 535
defined（定义） 515, 526
events and（事件） 526～527, 535

functionality（功能） 528～529
Golden Crustacean application（黄金蟹应用） 538～539
methods and（方法） 526～527
deleting directories（删除字典） 422
deployment packages（部署包）7, 35～37
dequeue, defined（出队，定义） 378
deserializing objects（逆串行化对象） 438～439, 442, 585
design patterns（设计模式） 536
dialog boxes（对话框）
CheckFileExists property（CheckFileExists属性） 420
CheckPathExists property（CheckPathExists属性） 420
as controls（控件） 420
creating（创建） 423
FileName property（FileName属性） 419, 421
Filter property（Filter属性） 420～421, 425
InitialDirectory property（InitialDirectory属性） 419～421
initialFolder property（initialFolder属性） 425
as objects（对象） 421
pop-up（弹出） 419
ShowDialog() method（ShowDialog()方法） 419～421, 423
Title property（Title属性） 419, 421, 425
dictionaries（字典）
beehive simulator example（蜂巢模拟系统示例） 556, 607
cleaning out（清除） 609
counting key-value pairs in（统计其中的键值对） 364
declaring（声明） 363
defined（定义） 363
examples （示例） 375
lists and（列表） 364
renderers and（表现工具） 606, 609
storing keys and values（存储键和值） 363
Dictionary class（Dictionary类）
Add() method（Add()方法） 363～364
beehive simulator example（蜂巢模拟系统示例） 556
ContainsKey() method（ContainsKey()方法） 363, 607
Count property（Count属性） 364
declaring（声明） 363
examples （示例） 363, 365
Keys property（Keys属性） 364
looking up values（查找值） 364
Remove() method（Remove()方法） 364
Directory class（Directory类）
CreateDirectory() method（CreateDirectory()方法） 422
Delete() method（Delete()方法） 422
functionality （功能） 424
GetFiles() method（GetFiles()方法） 422
DivideByZeroException, 465, 467, 472, 488, 498, 740
[DllImport] attribute（[DllImport]属性） 654
double backslash (\\)（双反斜线 (\\)） 423
double buffering（双缓冲） 634～639
double data type（double数据类型） 126～128, 131
downcasting（向下强制转换） 286～288
downloading logos（下载logo） 13
DragEventArgs delegate（DragEventArgs委托） 515


# E


else if statement（else if语句） 381, 434
empty classes（空类） 512
empty methods（空方法） 556
empty strings（空串） 62, 74, 475
encapsulation（封装）
automatic properties and（自动属性） 205
beehive simulator example（蜂巢模拟系统示例） 556, 561～562
classes and（类） 199～201, 459, 549
controlling access via（通过封装控制访问） 192～194
defined（定义） 189, 191, 306
dinner party application（宴会应用） 180～189
examples （示例） 202
hiding information via（通过封装隐藏信息） 197～200
OOP principles and（OOP原则） 306, 459
properties and（属性） 203
structs and（struct） 676
suggestions for（建议） 201
encoding data（编码数据）
defined（定义） 410, 423, 446
example（示例） 457
StreamWriter class（StreamWriter类） 449
Unicode support（Unicode支持） 446
UTF-8 encoding（编码） 457～458
Encoding.UTF8 method（Encoding.UTF8方法） 457
encrypting data（加密数据） 416
EndOfStreamException, 490
enqueue, defined（入队，定义） 378
entry points（入口点）
changing（改变） 54～55

creating（创建） 54
defined（定义） 52～53, 56
as static methods（作为静态方法） 243
enum data type（enum数据类型）
beehive simulator example（蜂巢模拟系统示例） 551～552
defined（定义） 329
examples （示例） 331～332, 381, 700
lists and（列表） 343
The Quest lab exercise（冒险者实验室练习） 394
representing numbers with names（用名表示数字） 330～331
storing data（存储数据） 343
enumerable objects（可枚举对象） 693, 753～755
enumeration（枚举） 328～329
(参见 enum data type（数据类型）)
enumerator lists（枚举项列表） 329
Enumerator objects（Enumerator对象）
Current property（Current属性） 355
defined（定义） 355
MoveNext() method（MoveNext()方法） 355
enumerators（枚举项） 329
Enum.Parse() method（Enum.Parse()方法） 438
Environment class（Environment类）
Exit() method（Exit()方法） 457
NewLine method（NewLine方法） 373, 379, 423
Error List window（错误列表窗口）
depicted（图示） 10, 47, 522
opening（打开） 9
program compilation and（程序编译） 56
escape sequences（转义序列）
@ symbol and（@符） 411～412, 423
defined（定义） 127
Environment.NewLine method（Environment.NewLine方法） 373
(参见 specific escape sequences)
event handler methods（事件处理方法）
adding automatically（自动增加） 194, 516～517
delegates and（委托） 526～527
event planning example（事件规划示例） 185, 222
implicit conversions（隐式转换） 522
subscribing classes and（订购类） 513
event handlers（事件处理程序）
adding（增加） 223, 515, 517
baseball simulation example（棒球模拟示例） 520～521
buttons and（按钮） 524～525, 585
chaining（串链） 535
Changed event handler（改变的事件处理程序） 430, 433
Click event handler（Click事件处理程序） 183, 403, 523, 525
for controls（控件） 221
creating automatically（自动创建） 516～517
debugging（调试） 474, 482
defined（定义） 183, 509, 511
functionality （功能） 511, 515
generic（通用） 522
hooking up（关联） 524～525
multiple（多个） 524～525
naming conventions（命名约定） 513
objects and（对象） 511
parameters（参数） 512
PictureBox control（PictureBox控件） 403
printing support（打印支持） 640～641
private（私有） 194
subscribing to events（订购事件） 513, 531～533
throwing exceptions（抛出异常） 524
timers and（定时器） 573
types of（类型） 515
event keyword（event关键字）, 512
event planning applications（事件规划应用）
birthday parties（生日聚会） 216～225, 252～256
dinner parties（宴会） 180～189, 252～256
EventArgs class（EventArgs类）
generic event handlers（通用事件处理程序） 522
inheritance（继承） 512, 515
EventHandler delegate（EventHandler委托）
event declaration（事件声明） 522
examples （示例） 527, 573
functionality （功能） 512, 515, 535
events（事件）
arguments for（参数） 512
baseball simulator application（棒球模拟系统应用） 509～514
callbacks and（回调） 536
chaining（串链） 524～525
checking for null value（检查null值） 535
connecting senders with receivers（连接发送者和接收者） 526
controls support（控件支持） 535
debugging（调试） 535

defined（定义） 163, 509
defining in classes（类中定义） 512
delegates and（委托） 526～527, 535
forms support（窗体支持） 523～525
Golden Crustacean application（黄金蟹应用） 533, 537～540
hooking up event handlers（关联事件处理程序） 524～525
IDE icons（IDE图标） 509
multiple handlers（多个处理程序） 524～525
naming conventions（命名约定） 514
raising（产生） 509～510, 514～515
subscribing to（订购） 509～510, 513, 531～533, 535
throwing exceptions（抛出异常） 514
triggering（触发） 510

Exception class（Exception类）
Data property（Data属性） 490
functionality （功能） 489, 495
inheritance（继承） 495, 497
Message property（Message属性） 472, 490
StackTrace property（StackTrace属性） 472, 490

exception handling（异常处理）
for arrays（数组） 470
beehive management system（蜂巢管理系统） 465～467, 491～492
beehive simulator example（蜂巢模拟系统示例） 585
BinaryFormatter object（BinaryFormatter对象） 477～478
catch-all（全能型） 484, 486, 488, 498
classes and（类） 491
comments and（注释） 498
constructors（构造函数） 483
exception objects（异常对象） 468
excuse manager program（借口管理程序） 464, 470～471, 474～477, 482～485, 501～502
finally block（finally块） 484～485, 501～502
functionality （功能） 486
IDisposable interface（IDisposable接口） 496～497, 500
MessageBox controls（MessageBox控件） 490
methods and（方法） 480, 491
multiple types of exceptions（多种异常类型）, 490
overloaded constructors（重载构造函数） 492
parameters（参数） 492
risky methods and（有风险的方法） 480
serialized files（串行化的文件） 478
suggestions（建议） 500
temporary solutions（临时解决方案） 499
tracking down exceptions（向下跟踪异常） 473
try/catch blocks（try/catch块） 479, 481～483, 498, 501～502
using statements（using语句） 495～497, 500
(参见 unhandled exceptions; specific exceptions)

exception objects（异常对象） 468, 472

exceptions（异常）
call stacks and（调用栈） 498
defined（定义） 468～469
events and（事件） 514
fixing（修正） 469, 471, 473, 499
preventing（预防） 473
rethrowing（重新抛出） 490
ToString() method（ToString()方法） 490
(参见 throwing exceptions)

exclamation point (!)（感叹号（!））475

excuse manager program（借口管理程序）
building（构建） 429～433, 459～460
exception handling（异常处理） 464, 470～471, 474～477, 482～485, 501～02

executing programs (参见program execution)

extension methods（扩展方法）
functionality （功能） 678～679
LINQ support（LINQ支持） 679, 689
namespace considerations（命名空间考虑） 680
static（静态） 680


# F


F5 key（F5按键） 483

F10 key（F10按键） 482

F11 key（F11按键） 474, 482, 529

F12 key（F12按键） 427

fields（字段，域）
adding to class diagrams（增加到类图） 100
adding to forms（增加到窗体） 114, 162
class skeletons and（类骨架） 546
controlling access to（控制访问） 192～194, 201
defined（定义） 100
defining in databases（数据库中定义） 22
inheriting（继承） 229, 233～234
interfaces and（接口） 273
lining up in forms（窗体中对齐） 32
masking（屏蔽） 208, 211
object state and（对象状态） 439
objects and（对象） 100
private（私有） 191, 193～194

properties and（属性） 100, 294
protected（保护） 253
Renderer class（Renderer类） 609
structs and（struct） 663
(参见 backing fields)
FIFO (first-in first-out)(FIFO（先进先出）) 377～378
File class（File类）
AppendAllText() method（AppendAllText()方法） 422
closing files automatically（自动关闭文件） 458
Create() method（Create()方法） 422, 445, 449, 451
Exist() method（Exist()方法） 422
FileInfo class and（FileInfo类） 458
function（函数） 424
functionality （功能） 422
GetLastAccessTime() method（GetLastAccessTime()方法） 422
GetLastWriteTime() method（GetLastWriteTime()方法） 422
Open() method（Open()方法） 491
OpenRead() method（OpenRead()方法） 422
OpenWrite() method（OpenWrite()方法） 422, 451
ReadAllBytes() method（ReadAllBytes()方法） 447, 458
ReadAllLines() method（ReadAllLines()方法） 458
ReadAllText() method（ReadAllText()方法） 425, 458
as static class（静态类） 422
WriteAllBytes() method（WriteAllBytes()方法） 458
WriteAllLines() method（WriteAllLines()方法） 458
WriteAllText() method（WriteAllText()方法） 425, 447, 458
File menu（File菜单）
Save All option（Save All选项） 15
Save option（Save选项） 15
FileInfo class（FileInfo类）
Exists() method（Exists()方法） 422
File class and（File类） 458
functionality （功能） 422
OpenRead() method（OpenRead()方法） 422
filenames（文件名）
@ prefix（@前缀） 411, 423
changing（改变） 11
exception handling（异常处理） 474～475
Length property（Length属性） 474
FileNotFoundException, 490～491, 497
files（文件）
allocating（分配） 427
appending text to（追加文本） 422
bundling with applications（与应用打包） 14
checking existence（检查是否存在） 422
closing（关闭） 458, 495
directory listing of（目录清单） 422
Dispose() method（Dispose()方法） 497
executable（可执行文件） 16
File class（File类） 422
getting information about（得到有关信息） 422
locked（加锁） 410～411
Name property（Name属性） 608
namespaces and（命名空间） 452
printing strings to（打印字符串） 423
reading from（读文件） 422
reading serialized（读串行化文件） 451～452
switching between（切换） 10
writing serialized（写串行化文件） 451～452
writing to（写文件） 411, 422
FileStream class（FileStream类）
Close() method（Close()方法） 410, 423
CryptoStream class and（CryptoStream类） 416
examples （示例） 416
functionality （功能） 409～410, 423, 458
StreamWriter class and（StreamWriter类） 411
writing binary data（写二进制数据） 449
finalizers（最终化方法）
defined（定义） 654
Dispose() method（Dispose()方法） 656～660
executing（执行）, 655
garbage collection and（垃圾回收） 654～657, 661
parameters and（参数） 661
serialization and（串行化） 659
stability and（稳定性） 658
throwing exceptions and（抛出异常） 661
finally block（finally块）
exception handling（异常处理） 501～502
finalizers as（最终化方法） 654
functionality （功能） 484～486
try/finally blocks（try/finally块） 497
first-in first-out (FIFO)（先进先出（FIFO）） 377～378
flickering（闪烁） 633
float data type（float数据类型） 127～128
FlowLayoutPanel control（FlowLayoutPanel控件）
Dock property（Dock属性） 425
FlowDirection property（FlowDirection属性） 425
FolderBrowserDialog dialog box（FolderBrowserDialog对话框） 419

Font object（Font对象） 623
fonts, creating（字体，创建） 623
for loops（for循环）
  beehive simulator example（蜂巢模拟系统示例） 563
  functionality （功能） 71
  writing（写） 65, 69
foreach loops（foreach循环）
  beehive simulator example（蜂巢模拟系统示例） 563, 610
  from clause and（from子句） 696
  defined（定义） 339～340
  examples （示例） 358, 380～381
  IEnumerable<T> interface（IEnumerable<T>接口） 355
  printing lists（打印列表） 354
Form Designer（窗体设计工具）
  adding PictureBox to forms（向窗体增加PictureBox） 48
  benefits（好处） 3
  expanding generated code（展开生成的代码） 48
  ToolStrip icon（ToolStrip图标） 585
Form1.cs file（Form1.cs文件）
  accessing（访问） 10
  changing filenames（改变文件名） 11
  creating（创建） 243
  functionality （功能） 8, 43
Form1.Designer.cs file（Form1.Designer.cs文件）
  adding picture controls（增加图片控件） 12
  changing filenames（改变文件名） 11
  Click event（Click事件） 523
  functionality （功能） 8, 43
Form1.resx file（Form1.resx文件） 14
FormatException, 465～466, 472
formatting strings（格式化字符串） 185
forms（窗体）
  adding buttons（增加按钮） 114, 357, 598, 602
  adding controls（增加控件） 12, 30～31, 590, 607
  adding fields（增加域） 114, 162
adding labels（增加标签） 114, 570～571
  adding PictureBox control（增加PictureBox控件） 48
  BackgroundImage property（BackgroundImage属性） 388, 617
  BackgroundImageLayout property（BackgroundImageLayout属性） 388, 608
  beehive simulator example（蜂巢模拟系统示例） 570
  building interactive（建立交互性） 114～116
  changing name of（改变标签名） 33
  Click event（Click事件） 622, 627
  ClientRectangle property（ClientRectangle属性） 622
  ClientSize property（ClientSize属性） 639
  connecting（连接） 612～613
  connecting to databases（连接到数据库） 17, 28～29
  Console Applications and（控制台应用） 244
  ControlBox property（ControlBox属性） 608
  CreateGraphics() method（CreateGraphics()方法） 618, 620～621, 632
  designing intuitive（设计直观）32
  displaying properties（显示属性） 10
  Dispose() method（Dispose()方法） 605
  drawing pictures（绘制图片） 622～623
  editing（编辑） 15
  event supported（支持的事件） 523～525
  FormBorderStyle property（FormBorderStyle属性） 160, 608
  getting back to（取回） 31
  Go Fish! game（Go Fish!游戏） 367～368
  hooking up event handlers（关联事件处理程序） 524～525
  InitializeComponent() method（InitializeComponent()方法） 421
  initializing controls（初始化控件） 208
  Invalidate() method（Invalidate()方法） 631, 636
  lining up fields and labels（对齐域和字段） 32
  linking（链接） 606
  Location property（Location属性） 608
  MaximizeBox property（MaximizeBox属性） 33
  maximizing（最大化） 33
  MinimizeBox property（MinimizeBox属性） 33
  Mouse event（Mouse事件） 612
  MouseClick event（MouseClick事件） 608
  MoveChildForms() method（MoveChildForms()方法） 612
  as objects（对象） 154～155
  OnDoubleClick event（OnDoubleClick事件） 514
  OnDoubleClick() method（OnDoubleClick()方法） 514
  overriding OnPaint method（覆盖OnPaint方法） 631
  Paint event（Paint事件） 628, 631
  private fields（私有字段） 191
  raising events（产生事件） 514
  redrawing（重绘） 628
  Refresh() method（Refresh()方法） 631
  removing controls（删除控件） 590
  repainting（重绘） 631
  Show() method（Show()方法） 606, 612～613
  StartPosition property（StartPosition属性） 608
  Text property（Text属性） 33

TextBox control filling up（TextBox控件填满） 425
tracking changes（跟踪变化） 430
Update() method（Update()方法） 631
frame rate, defined（帧速率，定义） 571
frames（帧）
defined（定义） 575
frame rates and（帧速率） 571
resizing images and（调整图像大小） 619
from clause（from子句）
functionality （功能） 692, 695～696
LINQ support（LINQ支持） 690

# G

GAC (Global Assembly Cache)（GAC （全局程序集缓存）743
garbage collection（垃圾回收）
defined（定义） 155
finalizers and（最终化方法） 654～657, 661
object references and（对象引用） 142, 147, 156
GC.Collect() method（GC.Collect()方法） 655, 657～658, 661
GDI+ (Graphics Device Interface)（GDI+（图形设备接口））620～622
generic collections（泛型集合）
built-in（内置） 377
defined（定义） 335, 340, 343
generic event handlers（通用事件处理程序） 522
geocaching（定向追踪） 198
get accessors（获取存取方法）
debugger and（调试工具） 476
defined（定义） 203
read-only property and（只读属性） 206, 209
this keyword（this关键字） and, 292
Global Assembly Cache (GAC)（全局程序集缓存（GAC））743
Go Fish! game（Go Fish!游戏） 366～376
Golden Crustacean application（黄金蟹应用） 533, 537～540
goto statements（goto语句） 739
GPS navigation system application（GPS导航系统应用） 86～92, 198
graph, defined（图形，定义） 441
Graphical User Interface (GUI)（图形用户界面（GUI）） 95
Graphics class（Graphics类）
DrawCircle() method（DrawCircle()方法） 621
DrawCurve() method（DrawCurve()方法） 623
DrawImage() method（DrawImage()方法） 618, 622, 627
DrawImageUnscaled() method（DrawImageUnscaled()方法） 638
DrawLine() method（DrawLine()方法） 620～621
DrawLines() method（DrawLines()方法） 623
DrawPolygon() method（DrawPolygon()方法） 623
DrawString() method（DrawString()方法） 621
FillCircle() method（FillCircle()方法） 621
FillPolygon() method（FillPolygon()方法） 623
FromImage() method（FromImage()方法） 618, 632
functionality （功能） 620
Paint events and（Paint事件） 628
printing support（打印支持） 640～645
resizing bitmaps（调整位图大小） 618
Graphics Device Interface (GDI+)（图形设备接口（GDI+））620～622
graphics files（图形文件） 14, 616～617
Griffiths, Ian, 765
group keyword（group关键字）, 699
GroupBox control（GroupBox控件） 177, 217
GUI (Graphical User Interface)（GUI（图形用户界面）） 95
GZipStream class（GZipStream类） 409

# H

handles, defined（句柄，定义） 660
heap（堆）
adding objects to（增加对象） 102
defined（定义） 102, 667
referencing objects（引用对象） 142
stack versus（栈与堆） 667～669, 676
Hebrew characters（希伯来字符） 446～447, 458
heisenbugs, defined（heisenbugs，定义） 476
hex dumps（十六进制转储） 453～455, 571
hexadecimal format（十六进制格式）
Character Map and（字符映射表） 446～447
converting to decimal（转换为十进制） 446, 450, 473
Debug toolbar and（Debug工具条） 473
hex dumps（十六进制转储） 453～455, 571
hiding methods（隐藏方法） 246～247, 249
hit points（点数） 386, 399, 404

house building application（房屋建设应用） 308～322

# I

IComparable<T> interface（IComparable<T>接口） 347
IComparer<T> interface（IComparer<T>接口）
  complex comparisons（复杂比较） 350
  creating instances（创建实例） 349
  examples （示例） 359
  sorting lists（列表排序） 346～348, 351
icons, bundling with applications（图标，与应用打包） 14
IDE
  Add Existing Item option（Add Existing Item选项） 263
  auto-generated code（自动生成的代码） 2, 11
  Basic Settings mode（Basic Settings模式） 473
  behind the scenes（在幕后） 14
  benefits using（使用的好处） 3
  building programs（构建程序） 34～35, 45
  changing generated code（改变生成的代码） 48～50
  defined（定义） 2, 44
  event icons（事件图标） 509
  Expert mode（Expert模式） 473
  functionality （功能） 8～11, 42～43
  Go To Definition feature（Go To Definition特性） 427
  New Project window（New Project窗口） 8
  snippets（片段） 47
  tools supported（支持的工具） 46～47, 473
  windows depicted（显示的窗口） 10
IDisposable interface（IDisposable接口）
  control class and（控件类） 603
  Dispose() method（Dispose()方法） 427～428, 496
  examples （示例） 355
  exception avoidance（异常避免） 496～497, 500
  finalizers and（最终化方法） 656, 658
  Font class and（Font类） 623
  functionality （功能） 427
  graphics support（图形支持） 632
IEnumerable<T> interface（IEnumerable<T>接口）
  collection initializers（集合初始化方法） 355
  collections and（集合） 689, 694
  creating lists（创建列表） 380
  creating queues（创建队列） 380
  creating stacks（创建栈） 380
  examples （示例） 359, 372
  extension methods and（扩展方法） 679
  foreach loops（foreach循环） 355
  LINQ support（LINQ支持） 689～690, 692, 707
  upcasting lists（向上强制转换列表） 356
IEquatable<T> interface（IEquatable<T>接口） 750
if statement（if语句）
  examples （示例） 113, 163, 350, 382
  testing if true（测试是否为true) 133
  for writing files（写文件） 434
if/else statements（if/else语句）
  code blocks and（代码块） 73
  examples （示例） 78, 434, 740
  functionality （功能） 67～68
  switch statement and（switch语句） 435
implicit conversion, defined（隐式转换，定义） 522
importing local resources（导入本地资源） 13
index (arrays)（索引（数组）） 150～151, 340
IndexOutOfRangeException, 468, 472
infinite loops（无限循环） 71
inheritance（继承）
  advantages of（好处） 226
  base classes and（基类） 226, 229～230, 234, 237, 239, 250～251
  beehive management system（蜂巢管理系统） 257～267, 271, 274
  birthday party application（生日聚会应用） 216～225, 252～256
  classes and（类） 225～227, 304, 590, 679
  CryptoStream class（CryptoStream类） 416
  defined（定义） 225, 227, 306
  dinner party application（宴会应用） 252～256
  EventArgs class（EventArgs类） 512, 515
  Exception class（Exception类） 495, 497
  exception objects（异常对象） 472
  extension methods and（扩展方法） 679
  grouping classes（类分组） 231
  hiding methods and（隐藏方法） 246～247, 249
  interface（接口） 281
  multiple（多个） 304
  from Object class（Object类） 445
  OOP principles and（OOP原则） 306
  overriding methods and（覆盖方法） 230, 232, 238, 248～249
  passing subclass instances（传递子类实例） 243
  PictureBox control（PictureBox控件） 600～601
  The Quest lab exercise（冒险者实验室练习） 400
  StreamReader class（StreamReader类） 415

structs and（struct） 663, 677
subclasses and（子类） 226, 229, 234, 237, 239, 250～251
zoo simulator application（动物园模拟系统应用） 228～234
initializing（初始化）
controls on forms（窗体上的控件） 208
lists（列表） 349
object initializers（对象初始化方法） 117, 206, 344, 602
private fields（私有字段） 207
instances（实例）
creating（创建） 101, 103, 217, 274
defined（定义） 94
IComparer<T> interface（IComparer<T>接口） 349
interface references and（接口引用） 279
List<T> objects（List<T>对象） 335
passing（传递） 243
polymorphism and（多态） 307
private fields and（私有字段） 191, 193～194
static methods and（静态方法） 99
tracking data（跟踪数据） 100
int data type（int数据类型）
casting as enum（强制转换为enum） 330
defined（定义） 20, 61, 126
examples （示例） 62
Int32 struct, 673
IntelliSense feature（智能提示特性）
accessing objects（访问对象） 287
color support（颜色支持） 621
event support（事件支持） 509, 514～515
functionality （功能） 47
object initializers（对象初始化方法） 117
overloaded methods（重载方法） 331
override methods（覆盖方法） 603
showing .NET interfaces（显示.NET接口） 273
interface inheritance（接口继承） 281
interface keyword（interface关键字）, 273
interface references（接口引用）
assigning instances（指定实例） 279
coffeemaker example（咖啡机示例） 287
IEnumerable<T> interface（IEnumerable<T>接口） 356
object references and（对象引用） 279, 294
interfaces（接口）
abstract classes and（抽象类） 296～297
beehive management system（蜂巢管理系统） 271～275, 279～283
on class diagrams（类图） 281
class requirements（类需求） 272
creating（创建） 276～277
defining（定义） 273
downcasting（向下强制转换） 287
enums and, 343
extending（扩展） 679
generic collections（泛型集合） 335, 340
implementing（实现） 274～275, 280～282, 288
is keyword（is关键字）, 280, 283
as keyword（as关键字）, 283, 286
List<T> class and（List<T>类） 340
methods and（方法） 272～273, 275, 281, 288, 294, 299
name considerations（命名考虑） 273
properties and（属性） 272～273, 275, 280–281
referencing（引用） 278～279
structs and（struct） 663
upcasting（向上强制转换） 285
zoo simulator application（动物园模拟系统应用） 288
internal access modifier（internal访问修饰符） 291
Invaders lab exercise（入侵者实验室练习） 713～733
InvalidCastException 465～466
InvalidOperationException 673
IOException 488, 490
is keyword（is关键字） 280, 283
iterators, defined（迭代器，定义） 69


# J


join clause（join子句）
examples （示例） 704, 710～711
functionality （功能） 703, 708～709
jump statements（jump语句） 738


# K


keys（键）
adding to collections（增加到集合） 365
counting key-value pairs（统计键值对） 364
defined（定义） 363

examples （示例） 375, 583, 700
getting list of（得到键列表） 364
primary（主键） 20～21
sequences and（序列） 708
storing（存储） 363


# L


lab exercises（实验室练习）
A Day at the Races（赛马日） 169～178
Invaders（入侵者） 713～733
The Quest（冒险游戏） 385～406
labels（标签）
adding to forms（增加到窗体） 114, 570～571
creating（创建） 598
event planning example（事件规划示例） 221
lining up in forms（窗体中对齐） 32
loops and（循环） 739
for methods（方法） 527
object references as（对象引用） 141, 156
for objects（对象） 527
updating（更新） 115, 571
lambda expressions（lambda表达式） 758～759
Language INtegrated Query (参见LINQ (Language INtegrated Query))
last-in first-out (LIFO)（后进先出（LIFO）） 377, 379
LIFO (last-in first-out)（LIFO（后进先出）） 377, 379
line break (\n)（换行（\n））
@ symbol and（@符） 411
casting（强制转换） 373
defined（定义） 423
examples （示例） 66, 89
functionality （功能） 15
in quotation marks（引号中） 127
linking forms（链接窗体） 606
LINQ (Language INtegrated Query)（LINQ（语言集成查询））
beehive simulator example（蜂巢模拟系统示例） 582～583
collection support（集合支持） 582～583, 689, 705
combining results into groups（结果合并到组）, 699
database support（数据库支持） 582～583, 705～708
defined（定义） 582
extension methods and（扩展方法） 679, 689
functionality （功能） 688
IEnumerable<T> interface（IEnumerable<T>接口） 689～690, 692, 707
query support（查询支持） 690～695
SQL and, 693, 705～708
versatility of（多面手） 694～695
XML documents and（XML文档） 582, 688, 762～763
LINQPad tool（LINQPad工具） 711
List<T> class（List<T>类）
adding to（增加到） 335
Add() method（Add()方法） 335, 340, 344
AddRange() method（AddRange()方法） 356
beehive simulator example（蜂巢模拟系统示例） 581～582
Clear() method（Clear()方法） 339
collection initializers（集合初始化方法） 344
Contains() method（Contains()方法） 339～340
converting arrays to lists（数组转换为列表） 343
Count property（Count属性） 340
creating instances（创建实例） 335
declaring（声明） 340
defined（定义） 335, 343
functionality （功能） 336
GetEnumerator() method（GetEnumerator()方法） 355
IndexOf() method（IndexOf()方法） 340
RemoveAt() method（RemoveAt()方法） 340
Remove() method（Remove()方法） 340
resizing dynamically（动态调整大小） 339～340
Sort() method（Sort()方法） 346～347
ListBox control（ListBox控件）
beehive simulator example（蜂巢模拟系统示例） 580
Dock property（Dock属性） 160
example（示例） 160
Font property（Font属性） 160
MultiColumn property（MultiColumn属性） 160
Name property（Name属性） 367
RedrawList() method（RedrawList ()方法） 381～382
SelectedIndex property（SelectedIndex属性） 359, 362
lists（列表）
arrays and（数组） 336, 338, 343
classes and（类） 343
data types and（数据类型） 343
dictionaries and（字典） 364
enum data type（enum数据类型） and, 343
of files in directories（目录中的文件） 422
foreach loops（foreach循环） 339～340
initializing（初始化） 349
practice exercise（实践练习） 358～362

printing contents（打印内容） 348, 354
queues and（队列） 377, 380
resizing dynamically（动态调整大小） 339
sorting（排序） 346～348, 351
stacks and（栈） 380
storing data（存储数据） 335, 343
ToArray() method（ToArray()方法） 343
upcasting（向上强制转换） 356
value types and（值类型） 343
literals（字面量）
@ symbol and（@符） 411
data types supporting（数据类型支持） 127
suffixes supported（支持的后缀） 156, 182, 185
logical operators（逻辑操作符） 68
logos, downloading（logo，下载） 13
long data type（long数据类型） 126
loops（循环）
adding to programs（增加到程序） 65, 69
comparing binary files（比较二进制文件） 451
conditional tests and（条件测试） 73
defined（定义） 65
examples （示例） 455
IEnumerable<T> interface（IEnumerable<T>接口） 355
infinite（无限） 71
labels and（标签） 739
nested（嵌套） 77
reading/writing lines（读/写行） 415

# M

Main() method（Main()方法）
Console Application and（控制台应用） 244～245
as entry point（作为入口点） 52～53, 56, 456
keeping program open（保持程序打开） 345
managed resources（托管资源） 654
masking fields（屏蔽字段） 208, 211
Math class（Math类）
Abs() method（Abs()方法） 551
PI member（PI成员） 66
Maximize button（Maximize按钮） 33
members class（成员类） 291～292
memory（内存）
allocating（分配） 427
arrays and（数组） 343
heap and（堆） 102, 142, 667～669
stack and（栈） 128, 667～669
value types and（值类型） 156
variables and（变量） 128
MemoryStream class（MemoryStream类） 409
MessageBox controls（MessageBox控件）
exception handling（异常处理） 490
finalizers and（最终化方法） 656
Show() method（Show()方法） 54, 431
metadata（元数据） 443
methods（方法）
abstract（抽象） 296, 299
anonymous（匿名） 758～759
arguments（参数） 133
callback（回调） 532～534
calling other methods（调用其他方法） 154, 183, 477
chaining onto events（链至事件） 524～525
class diagrams and（类图） 91
class skeletons and（类骨架） 546
classes and（类） 50, 89
as code blocks（代码块） 15
concrete（具体） 296, 299
constructors and（构造函数） 207
controlling access to（控制访问） 192～194, 201
curly brackets and（大括号） 66
declaring（声明） 52
declaring variables inside（在内部声明变量） 116
defined（定义） 15, 50～51
delegates and（委托） 526～527
empty（空） 556
enums and, 343
event handlers（事件处理程序） 183, 513
examples （示例） 53
exception handling and（异常处理） 480, 491
extension（扩展） 678～679, 689
extracting（抽取） 756
get accessors（获取存取方法） 203, 206, 209
hiding（隐藏） 246～247, 249
inheriting（继承） 229, 233～234
interface requirements（接口需求） 272～273, 275, 281, 288, 294, 299
labeling（标示） 527
name considerations（命名考虑） 104～105, 118
objects and（对象） 93, 100
overloaded（重载） 253, 331, 357
parameters（参数） 51, 88, 133, 211, 672

passing subclass instances（传递子类实例） 243
polymorphism and（多态） 307
properties and（属性） 209
raising events（产生事件） 514
recursion（递归） 669
return types（返回类型） 89
return values（返回值） 51, 88, 207, 209, 394
set accessors（设置存取方法） 203, 209
sharing with other classes（与其他类共享） 90
static（静态） 99, 243, 448, 458, 676
structs and（struct） 664
updating labels with（用来更新标签） 115
virtual（虚） 238, 243～244, 248～249, 353
(参见 overriding methods)

Minimize button（Minimize按钮） 33

modifiers, defined（修饰符，定义） 670

multiple event handlers（多个事件处理程序） 524～525

multiple inheritance（多重继承） 304


# N


named arguments（命名参数） 672

namespaces（命名空间）
assemblies and（程序集） 742～745
classes and（类） 59
defined（定义） 44, 50, 53
examples （示例） 53
extension methods and（扩展方法） 680
files and（文件） 452
not recommended for use（不建议使用） 73

naming conventions（命名约定）
camelCase, 211
for event handlers（事件处理程序） 513
for events（事件） 514
PascalCase, 211

navigation system application（导航系统应用） 86～92, 198

nested loops（嵌套循环） 77

nested using statements（嵌套using语句） 496

.NET database objects（.NET数据库对象） 6, 28～29

.NET Framework（.NET框架）
generic collections（泛型集合） 377
overview（概述） 44
System namespace（System命名空间） 73
using statement（using语句） 50

.NET visual objects（.NET虚对象） 6, 12～13

NetworkStream class（NetworkStream类） 409

new keyword/statement（new关键字/语句）
collection initializers and（集合初始化方法） 344
creating array objects（创建数组对象） 150
creating classes（创建类） 207
creating objects（创建对象） 92～93
debugging programs and（调试程序） 482
hiding methods and（隐藏方法） 247
implicit conversion and（隐式转换） 522
overriding methods and（覆盖方法） 249
passing parameters to（传递参数） 207

non-visual controls（不可见控件） 161, 420

null keyword（null关键字）, 155, 465

nullable types（nullable类型） 673～674

Nullable<T> struct, 673

NullReferenceException, 116, 465, 467, 535

NumericUpDown control（NumericUpDown控件）
baseball simulator example（棒球模拟系统示 例） 519
event planning example（事件规划示例） 183, 221～222
GPS navigation system example（GPS导航系统示例） 90
Value property（Value属性） 519
ValueChanged event（ValueChanged事件） 513


# O


Object class（Object类） 353, 445

object data type（object数据类型） 127

object declaration (参见declaration)

object initializers（对象初始化方法）
beehive simulator example（蜂巢模拟系统示例） 602
collection initializers and（集合初始化方法） 344
functionality （功能） 117
initializing properly（适当初始化） 206

object oriented programming (OOP)（面向对象编程（OOP）） 306, 459

object references（对象引用）
arrays of（数组） 151
callback techniques（回调技术） 535
Controls collection and（Controls集合） 591
defined（定义） 156
examples （示例） 287

garbage collection and（垃圾回收） 142, 147, 156
interface references and（接口引用） 279, 294
as labels（标签） 141, 156
multiple（多个） 144, 149
reference variables and（引用变量） 140～141
Object Relational Designer（对象关系设计工具） 707～708
objects（对象）
accessing with IntelliSense（利用智能提示访问） 287
adding to heap（增加到堆） 102
allocating（分配） 427
base class for（基类）353
as black boxes（黑盒） 199～200
built-in（内置） 419
callback technique（回调技术） 534～535
chaining（串链） 515
comparing（比较） 347
controls as（控件） 591
converting to strings（转换为字符串） 353～354
creating（创建） 92～94
dead（死） 655
defined（定义） 92
deserializing（串行化） 438～439, 442, 585
downcasting（向下强制转换） 286
enqueuing/dequeuing（入队/出队）, 378
enumerable（可枚举） 693
Equals() method（Equals()方法） 750～752
event handling （事件处理） 511
exceptions as（异常） 469
fields and（字段） 100
GPS navigation system application（GPS导航系统应用） 86～92, 198
initializing（初始化） 117, 206
instantiating（实例化） 741
interacting with buttons（与按钮交互） 115
labeling（标示） 527
methods and（方法） 100
misusing（误用） 190
multiple using statements（多个using语句） 428
NullReferenceException, 116
ordering/sorting（排序） 346～348
polymorphism and（多态） 307
populating classes with（填充类） 365
private fields and（私有字段） 191, 193
reference variables and（引用变量） 154
removing（删除） 340
serializing（串行化） 438～442
structs and（struct） 663
subscribing to events（订购事件） 513, 531～533, 535
ToString() method（ToString()方法） 132, 353～354
upcasting（向上强制转换） 285, 512
value types versus（值类型） 664
variables and（变量） 139
on...equals clause（on...equals子句） 708
OOP (object oriented programming)（OOP（面向对象编程））306, 459
OpenFileDialog control（OpenFileDialog控件） 420, 425, 605
OpenFileDialog dialog box（OpenFileDialog对话框） 419
OpenFileDialog object（OpenFileDialog对象） 421, 423
operators（操作符）
comparison（比较） 70
compound（复合） 138
conditional（条件） 68, 739
defined（定义） 62
logical（逻辑） 68
orderby clause（orderby子句）
beehive simulator example（蜂巢模拟系统示例） 699
functionality （功能） 692, 695～696, 703
LINQ support（LINQ支持） 690
Output window, accessing（Output窗口，访问） 204
OverflowException, 465～466, 472
overloaded constructors（重载构造函数）
Bitmap class and（Bitmap类） 619
examples （示例） 359
exception handling and（异常处理） 492
passing collections as parameters（传递集合作为参数） 380
StreamWriter class（StreamWriter类） 411
overloaded methods（重载方法） 253, 331, 357
override keyword（override关键字）, 238, 244, 248～249
overriding methods（覆盖方法）
converting objects to strings（对象转换为字符串） 353～354
examples （示例） 232, 248
functionality （功能） 230, 238, 244, 248～249
hiding methods versus（隐藏方法） 246, 249
Intellisense and（智能提示） 603


# P


parameters（参数）
on class diagrams（类图） 394
for constructors（构造函数） 207～208

defined（定义） 669
event handlers（事件处理程序） 512
exception handling（异常处理） 492
finalizers and（最终化方法） 661
for methods（方法） 51, 88, 133, 211, 670, 672
for overloaded constructors（重载构造函数） 359, 380
for overloaded methods（重载方法） 357
passing command-line arguments（传递命令行参数） 457
passing to new statement（传递到 new语句） 207
setting default values（设置默认值） 672
parentheses(), 184
partial classes（部分类） 73
PascalCase, 211
pass by reference（按引用传递） 671
pass by value（按值传递）671, 677
patterns, defined（模式，定义） 534～536
Pen object（Pen对象） 622, 632
performance considerations（性能考虑） 615～617
PictureBox control（PictureBox控件）
adding pictures（增加图片） 12
adding to forms（增加到窗体） 48
BackColor property（BackColor属性） 389
BackgroundImage property（BackgroundImage属性） 608
beehive simulator example（蜂巢模拟系统示例） 609
BorderStyle property（BorderStyle属性） 403
editing（编辑） 15
event handlers（事件处理程序） 403
inheritance（继承） 600～601
performance issues（性能问题） 615
The Quest lab exercise（冒险者实验室练习） 388～389, 403～405
resource files and（资源文件） 14
setting to Zoom mode（设置为Zoom模式） 13
Size property（Size属性） 13
user controls and（用户控件） 605
Visible property（Visible属性） 405
Point struct
beehive simulator example（蜂巢模拟系统示例） 607
drawing graphics（绘制图形） 621
functionality （功能） 565, 665
namespace support（命名空间支持） 547～548
polymorphism（多态） 306～307, 335
popping out of stacks（弹出栈） 379
pop-up dialog boxes（弹出对话框） 419
primary keys（主键） 20～21
PrintDialog object（PrintDialog对象） 641
PrintDocument class（类）
functionality （功能） 641
Print() method（Print()方法） 640
PrintPage event（PrintPage事件） 640～642
printing（打印）
graphics and（图形） 640～645
lists（列表） 348, 354
strings to files（字符串打印到文件） 423
PrintPreviewDialog object（PrintPreviewDialog对象） 641
private access modifier（private访问修饰符） 291, 294, 536
private fields（私有字段）
declaring（声明） 191
initializing（初始化） 207
instances and（实例） 191, 193～194
program builds（程序构建）
changing code and（修改代码） 601
with classes（类） 89～91
IDE process（IDE进程） 34～35, 45
program compilation（程序编译）
assigning variable values and（赋变量值） 130
comments and（注释） 63
Error List window and（错误列表窗口） 56
IDE process（IDE进程） 34, 45
invalid arguments error（非法参数错误） 133
just-in-time compiler（即时编译器） 745
var keyword and（var关键字） 690
program execution（程序执行）
changing entry point（改变入口点） 54
entry point for（入口点） 52～53
event handlers and（事件处理程序） 509
IDE process（IDE进程） 34, 45
stopping（停止） 34
unhandled exceptions and（未处理异常） 476
Program.cs file（Program.cs文件）
accessing（访问） 10
changing filenames（改变文件名） 11
creating（创建） 243
entry point for（入口点） 52
functionality （功能） 8, 43
Main() method（Main()方法） 244～245, 345, 456
project files (.csproj)（工程文件（.csproj）） 44
Project menu（Project菜单） 35

projects（工程）
- adding classes（增加类） 90
- adding databases（增加数据库） 18
- creating（创建） 8, 11, 113
- saving（保存） 8, 11
- storing data（存储数据） 27

properties（属性）
- class skeletons and（类骨架） 546
- defined（定义） 89, 203
- encapsulation and（封装） 203
- fields and（字段） 100, 294
- hovering over（停留在上面） 476
- inheriting（继承） 229, 233～234
- interface requirements（接口需求） 272～273, 275, 280～281
- methods and（方法） 209
- structs and（struct） 663
- (参见 automatic properties; read-only properties)

Properties window（属性窗口）
- accessing（访问） 33
- changing filenames（改变文件名） 11
- changing view（改变视图） 163
- displaying for forms（显示窗体的...） 10
- Events button/page（事件按钮/页） 223, 523, 572, 612, 622, 628
- functionality （功能） 42

protected access modifier（protected访问修饰符） 291, 294

protected fields（保护字段） 253

public access modifier（public访问修饰符） 291, 294, 512

publish folder（发布文件夹） 36

Publish Wizard（发布向导） 35

pushing onto stacks（压入栈） 379

## Q

queries（查询）
- combining values into groups（值合并到组） 700
- defined（定义） 19
- editing（编辑） 711
- joining data results（连接数据结果） 703～704
- LINQ support（LINQ支持） 690～695, 763

The Quest lab exercise（冒险者实验室练习） 385～406

question mark (?)（问号（?）） 673

queues（队列）
- Clear() method（Clear()方法） 378
- copying to stacks（复制到栈） 380
- Count property（Count属性） 378
- creating（创建） 378
- defined（定义） 377
- Dequeue() method（Dequeue()方法） 378
- exercises（练习） 381–382
- FIFO support（FIFO支持） 377～378
- lists and（列表） 377, 380
- Peek() method（Peek()方法） 378
- stacks and（栈） 377, 379～380

## R

racetrack simulator application（赛场模拟系统应用） 169～178

raising events（产生事件）
- baseball simulator example（棒球模拟系统示例） 514
- defined（定义） 509～510
- this keyword and（this关键字） 515

Random class（Random类） 331, 559

random numbers（随机数） 194, 559, 563

range variables（范围变量） 690

reading data（读数据）
- BinaryReader class（BinaryReader类） 450
- bytes from streams（从流读字节） 456～457
- excuse manager program（借口管理程序） 429～433
- File class（File类） 422
- FileStream class（FileStream类） 410
- serialized files（串行化文件） 451～452
- Stream class（Stream类） 408～409
- StreamReader class（StreamReader类） 415
- switch statement（switch语句） 436～437

read-only properties（只读属性）
- adding（增加） 205～206
- beehive simulator example（蜂巢模拟系统示例） 546, 551
- event arguments and（事件参数） 520
- functionality （功能） 549
- get accessors and（获取存取方法） 209

Ready Bake Code（现成代码） 583, 688

real numbers（实数） 126

record IDs（记录ID） 21

Rectangle struct
- beehive simulator example（蜂巢模拟系统示例） 638
- functionality （功能） 676

The Quest lab exercise（冒险者实验室练习） 392, 403
recursion, defined（递归，定义） 669
ref keyword（ref关键字）, 671
refactoring（重构） 756～757
reference types（引用类型）
== operator and （==操作符） 750
value types and（值类型） 664～666
reference variables（引用变量）
arrays and（数组） 151
code example（代码示例） 526
declaring（声明） 150
defined（定义） 140～141
delegate types and（委托类型） 527
objects and（对象） 154
Renderer class（Renderer类）
beehive simulator example（蜂巢模拟系统示例） 594～595, 607, 609～611, 637
building（构建） 609～611
functionality （功能） 594
ResizeImage() method（ResizeImage()方法） 618
reserved words (C#)（保留字(C#)） 156, 164
resizing（调整大小）
backing fields（后备字段） 219
images（图像） 618, 632
lists dynamically（动态调整列表大小） 339
Resource Designer（资源设计工具） 608
resource files (.resx)（资源文件(.resx)） 46
resources（资源）
allocated（已分配） 427
defined（定义） 14
finalizers and（最终化方法） 654
managed（托管） 654
storing as Bitmap objects（存储为Bitmap对象） 619
unmanaged（非托管） 654
rethrowing exceptions（重新抛出异常） 490
return character (\r)（回车符(\r)）, 373, 423
return statement（return语句） 51, 88～89, 753～755
return types（返回类型） 88～89, 208
return values (methods)（返回值（方法））
= operator and （=操作符） 739
on class diagrams（类图） 394
constructors and（构造函数） 207, 209
defined（定义） 51, 88
delegates and（委托） 527
multiple（多个） 670
robust programs（健壮的程序） 478, 674
RSS feeds（RSS提要） 763

# S

Save As... dialog box（Save As...对话框） 421
SaveFileDialog control（SaveFileDialog控件） 425
SaveFileDialog object（SaveFileDialog对象） 421
sbyte data type（sbyte数据类型） 126
sealed (access) modifier（sealed（访问）修饰符） 291, 678～679
seeds（种子） 194, 559
select clause（select子句）
anonymous types（匿名类型） 703, 709
beehive simulator example（蜂巢模拟系统示例） 699
functionality （功能） 695～696
LINQ support（LINQ支持） 690
Select Resource dialog box（Select Resource对话框） 13
Sells, Chris, 765
semicolon (;)（分号（;））
interface requirements（接口需求） 273
statements and（语句） 47, 66, 73
void return type and（void返回类型） 88
sequences（序列）
CurrentState property（CurrentState属性） 699
defined（定义） 695
examples （示例） 700
keys and（键） 708
[Serializable] attribute（[Serializable]属性） 443～444
serialization（串行化）
beehive simulator example（蜂巢模拟系统示例） 585
classes（类） 443
DataContractSerializer class（DataContractSerializer类） 760～761
exception handling（异常处理） 478
finalizers and（最终化方法） 659
objects（对象） 438～442
reading/writing files manually（手动读/写文件） 451～452
SerializationException, 477～478, 484

Server Explorer window（Server Explorer窗口） 18

set accessors（设置存取方法）

callback methods and（回调方法） 534

defined（定义） 203

this keyword and,（this关键字）292

value parameter（value参数） 203, 209

setup programs（建立程序） 36

short data type（short数据类型） 126

single quotes ( ')（单引号( ')） 127

Size struct, 642, 676

snippets (IDE)（片段（IDE）） 47

Solution Explorer window（Solution Explorer 窗口）

adding SQL databases to projects（将SQL数据库增加到工程） 18

changing filenames（改变文件名） 11

displaying files（显示文件） 14

functionality （功能） 46

opening designer code（打开设计工具代码） 48

project files（工程文件） 8

switching between files（切换文件） 10

viewing databases（查看数据库） 18

solution files (.sln)（解决方案文件(.sln)） 44

sorting lists（列表排序） 346～348, 351

source code files（源代码文件） 44

SQL (Structured Query Language)（SQL（结构化查询语言）

defined（定义） 19

LINQ and, 693, 705～708

SQL databases（SQL数据库） 18～19, 34

SQL Server Compact

database files（数据库文件） 18, 27

installing（安装） 36

LINQ support（LINQ支持） 709

SqlMetal.exe tool（SqlMetal.exe工具） 706, 709

stacks（栈）

creating（创建） 379～380

defined（定义） 128, 377, 667

examples （示例） 381～382, 669

heaps versus（堆） 667～669, 676

LIFO support（LIFO支持） 376, 379

lists and（列表） 380

Peek() method（Peek()方法） 379

Pop() method（Pop()方法） 379

popping out of（弹出） 379

pushing onto（压入） 379

Push() method（Push()方法） 379

queues and（队列） 377, 379～380

value types and（值类型） 128

statements（语句）

adding（增加） 66

classes and（类） 73

defined（定义） 19, 51, 53, 73

grouping into blocks（分组到块） 56

polymorphism and（多态） 307

semicolons and（分号） 47, 66, 73

structs and（struct） 664

throwing exceptions（抛出异常） 482, 495

(参见 specific statements)

states (object)（状态（对象））

beehive simulator example（蜂巢模拟系统示例） 552～553, 583

overview（概述） 439～440

static classes（静态类） 422, 678

static extension methods（静态扩展方法） 680

static methods（静态方法）

for arrays（数组） 448

closing files automatically（自动关闭文件） 458

defined（定义） 99

entry points as（入口点） 243

extension methods and（扩展方法） 678

instances and（实例） 99

StatusStrip control（StatusStrip控件）

adding（增加） 160～161, 570

functionality （功能） 570

Name property（Name属性） 161

SizingGrip property（SizingGrip属性） 161

Text property（Text属性） 161

Stop Debugging button（Stop Debugging按钮） 16

stored procedures（存储过程） 19

Stream class（Stream类）

as abstract class（抽象类） 409

Dispose() method（Dispose()方法） 428

functionality （功能） 408～409

Read() method（Read()方法） 456～457

subclasses supported（支持的子类） 409～410

StreamReader class（StreamReader类）

Close() method（Close()方法） 415

converting strings to byte arrays（字符串转换为字节数组） 423

EndOfStream property（EndOfStream属性） 415

FileStream class and（FileStream类） 423
functionality （功能） 458
inheritance（继承） 415
ReadBlock() method（ReadBlock()方法） 455
ReadLine() method（ReadLine()方法）415

streams（流）
built-in classes（内置类） 409
chaining（串链） 416
closing（关闭） 410～411, 415, 423, 428
constructors and（构造函数） 415
defined（定义） 408
deserializing objects（逆串行化对象） 442
Dispose() method（Dispose()方法） 428, 497
functionality （功能） 408～409
hex dumps（十六进制转储） 453～454
Length field（Length字段） 451
multiple（多个） 416
opening（打开） 449
reading bytes from（从中读取字节） 456～457
Read() method（Read()方法） 409
Seek() method（Seek()方法） 409
serializing objects（串行化对象） 438, 442
using statements and（using语句） 428
Write() method（Write()方法） 409～410

StreamWriter class（StreamWriter类）
Close() method（Close()方法） 411～412, 415, 433
converting strings to byte arrays（字符串转换为数组） 423
encoding data（编码数据） 449
examples （示例） 412, 415
FileStream class and（FileStream类） 423
functionality （功能） 411, 458
WriteLine() method（WriteLine()方法） 411～412
Write() method（Write()方法） 411

String class（String类）
Format() method（Format()方法） 353, 454～455, 457, 571
IsNullOrEmpty() method（IsNullOrEmpty()方法） 258

string data type（string数据类型）
defined（定义） 61, 126
memory considerations（内存考虑） 128
Unicode standard（Unicode标准） 447

StringBuilder class（StringBuilder类）
AppendFormat() method（AppendFormat()方法） 741
AppendLine() method（AppendLine()方法） 741
Append() method（Append()方法） 741

strings（字符串）
concatenating（连接） 132, 353
converting byte arrays to（字节数据转换为） 457
converting char to（字符转换为） 455
converting objects to（对象转换为） 353～354
converting to byte arrays（转换为字节数组） 423
data storage and（数据存储） 328
dictionaries and（字典） 363
empty（空） 62, 74, 475
extending（扩展） 680
formatting（格式化） 185
Length property（Length属性） 89
printing to files（打印到文件） 423
Split() method（Split()方法） 437
Substring() method（Substring()方法） 455
ToLower() method（ToLower()方法） 381～382
ToString() method（ToString()方法） 132, 185, 330, 353, 381～382
Unicode standard（Unicode标准） 447

struct value type（struct值类型）
boxing in wrappers（装箱到包装器中） 668, 676
classes and（类） 676
creating（创建） 665
defined（定义） 663～665
encapsulation and（封装） 676
inheritance and（继承） 663, 677
ToString() method（ToString()方法） 663

Structured Query Language (SQL)（结构化查询语言（SQL））
defined（定义） 19
LINQ and, 693, 705～708

subclasses（子类）
access modifiers and（访问修饰符） 291
base classes and（基类） 233, 239, 250
constructors and（构造函数） 251
grouping classes（类成组） 231
hiding methods（隐藏方法） 246～247
inheritance and（继承） 226, 229, 234, 237, 239, 250～251
overriding methods（覆盖方法） 230, 238, 246
passing instances（传递实例） 243
protected fields（保护字段） 253
upcasting from（由子类向上强制转换列） 285

subscribing to events（订购事件）
chaining and（串链） 515
defined（定义） 509～510
event handler methods and（事件处理方法） 513

getting notified（得到通知） 511
objects（对象） 531～533, 535
superclasses（超类） 246～247, 356
switch statement（switch语句）
beehive simulator example（蜂巢模拟系统示例） 551, 617
deck of cards example（扑克牌示例） 436～437
functionality （功能） 435
SSystem namespace（System命名空间） 73
System.Collections namespace（System.Collections命名空间） 343
System.Collections.Generic namespace（System.Collections.Generic命名空间） 377
System.Data namespace（System.Data命名空间） 73
System.Drawing namespace（System.Drawing命名空间） 547～548, 611, 620
System.IO namespace（System.IO命名空间） 73, 410, 412
System.Linq namespace（System.Linq命名空间） 688, 694
System.Runtime.Serialization namespace（System.Runtime.Serialization命名空间） 484
System.Threading namespace（System.Threading命名空间） 476
System.Windows.Forms namespace（System.Windows.Forms命名空间） 54, 601～602, 611

# T

tab (\t)（制表符（\t）） 127, 411, 423
TabControl control（TabControl控件） 217, 221
TableLayoutPanel control（TableLayoutPanel控件）
Columns property（Columns属性） 425
Dock property（Dock属性） 425
labels and（标签） 570
The Quest lab exercise（冒险者实验室练习） 388
Rows property（Rows属性） 425
tables（表）
adding columns（增加列） 20, 22～24
binding controls to（绑定控件） 31
creating（创建） 20～21, 25
creating primary keys（创建主键） 20
defined（定义） 19
selecting（选择） 30
SQL support（SQL支持） 693
(参见 databases)
tabs（制表符）
TabCollection property（TabCollection属性） 221
TabPages property（TabPages属性） 221
Text property（Text属性） 221
Take() command（Take()命令） 696
testing（测试）
beehive simulator example（蜂巢模拟系统示例） 577～579, 584, 614
conditional（条件） 67～70, 73
deployment（部署） 37
installation（安装） 37
programs（程序） 34
values（值） 436
text（文本）
adding fonts（增加字体） 623
storing（存储） 447
writing to files（写至文件） 411
TextBox control（TextBox控件）
Changed event handler（改变的事件处理程序） 430
event planning example（事件规划示例） 223
filling up forms（填充窗体） 425
Name property（Name属性） 367
ReadOnly property（ReadOnly属性） 367
ScrollToCaret() method（ScrollToCaret()方法） 368
SelectionStart property（SelectionStart属性） 368
Text property（Text属性） 90, 221
TextReader class（TextReader类） 415
this keyword（this关键字）
defined（定义） 154, 156
examples （示例） 292
extension methods and（扩展方法） 680
masking fields and（屏蔽字段） 208, 211
raising events（产生事件） 515
Thread.Sleep() method（Thread.Sleep()方法） 476
throwing exceptions（抛出异常）
beehive simulator example（蜂巢模拟系统示例） 565
classes（类） 491～492
defined（定义） 490
event handlers（事件处理程序） 524
events（事件） 514
finalizers and（最终化方法） 661
statements（语句） 482, 495
tilde (~)（波浪线（~）） 654
Timer control（Timer控件）

adding（增加） 160～161, 570
beehive simulator example（蜂巢模拟系统示例） 574～576, 601, 636
Dispose() method（Dispose()方法） 603
Enabled property（Enabled属性） 572
event handlers and（事件处理程序） 573
functionality （功能） 572
Interval property（Interval属性） 161
Start() method（Start()方法） 572
Stop() method（Stop()方法） 572
Tick event（Tick事件） 572～573, 605, 640

TimeSpan class（TimeSpan类）
Days property（Days属性） 574
functionality （功能） 571, 574
Hours property（Hours属性） 574
Milliseconds property（Milliseconds属性） 574
Seconds property（Seconds属性） 574

title bars（标题栏） 33

ToArray() command（ToArray()命令） 695

ToDictionary() command（ToDictionary()命令） 695

ToList() command（ToList()命令） 695

toolbars, adding（工具条，增加） 570

Toolbox window（Toolbox窗口）
adding controls（增加控件） 604
creating controls（创建控件） 590
depicted（图示） 10
displaying controls in（显示控件） 160, 599
non-visual controls（不可见控件） 420
opening（打开） 9
PictureBox control（PictureBox控件） 12

ToolStrip control（ToolStrip控件）
CanOverflow property（CanOverflow属性） 585
functionality （功能） 570
GripStyle property（GripStyle属性） 585
insertion feature（插入特性） 585

TrackBar control（TrackBar控件） 629

transparency（透明性） 615, 627

troubleshooting (参见exception handling)

try/catch blocks（try/catch 块）
debugger and（调试工具） 482～483
exception handling（异常处理） 479, 481～483, 498, 501–502
finally block and（finally块） 484～485
function（函数） 495
functionality （功能） 479, 481, 486
multiple exceptions and（多个异常） 490
variable names and（变量名） 489～490

try/finally blocks（try/finally块） 497

turn-based systems（基于回合的系统） 386, 561, 575

type arguments（type参数） 340

Type.GetType() method（Type.GetType()方法） 749

types (参见data types)

typing game building（打字游戏构建） 160～163


# U


uint data type（uint数据类型） 126

ulong data type（ulong数据类型） 126

unhandled exceptions（未处理异常）
adding informative messages（增加信息型消息） 471
defined（定义） 486, 495
excuse manager example（借口管理系统示例） 464, 470
program execution and（程序执行） 476
viewing（查看） 468, 476～477

Unicode Consortium, 446

Unicode standard（Unicode标准）
character examples （字符示例） 448
defined（定义） 446
encoding functionality （编码功能） 457～458
Hebrew letter examples （希伯来字符示例） 446～447
viewing values in Watch window（监视窗口中查看值） 373

unmanaged resources（非托管资源） 654

upcasting（向上强制转换）
defined（定义） 285
downcasting versus（向下强制转换） 288
interfaces（接口） 285
lists（列表） 356
objects（对象） 285, 512

user controls（用户控件）
building（构建） 629～630
DoubleBuffered property（DoubleBuffered属性） 635
functionality （功能） 604～605
hosting other controls（包含其他控件） 632
Paint event（Paint事件） 629～630
PictureBox controls and（PictureBox控件） 605

user interfaces（用户界面）
beehive simulator example（蜂巢模拟系统示例） 592
developing（开发） 3, 12～13
user needs, identifying（用户需求，明确） 5
ushort data type（ushort数据类型） 126
using statements（using语句）
allocated resources（分配的资源） 427
auto-generated（自动生成的） 50, 54
building overloaded methods（构建重载方法） 357
classes and（类） 73, 496
closing streams（关闭流） 428
Dispose() method（Dispose()方法） 495, 497, 656
exception handling and（异常处理） 495～497, 500
functionality （功能） 53, 428
graphics support（图形支持) 632
multiple（多个） 428
nested(嵌套) 496
serializing classes（串行化类） 443
UTF-8 encoding（UTF-8编码） 457～458

# V

value parameter（value参数） 203, 209
value types(值类型)
== operator and （==操作符） 750
enums and, 343
as keyword and, （as关键字）668, 677
lists and（列表） 343
memory considerations（内存考虑） 156
objects versus（对象） 664
question mark and（问号） 673
reference types and（引用类型） 664～666
stack and（栈） 128
structs, 663–665
ToString() method（ToString()方法） 132
TryParse() method（TryParse()方法） 671
values（值）
combining into groups（结合到组） 700
counting key-value pairs（统计键值对） 364
defined（定义） 363
returning multiple（返回多个） 670
setting default（设置默认值） 672
storing（存储） 363
testing（测试） 436
Unicode, 373, 446
ValueType class（类） 663
var关键字）, 690, 708
variables（变量）
++ operator and （++操作符） 739
adding watch in debugger（调试工具中增加监视） 64
arrays and（数组） 150～151
assigning values（赋值） 61, 130
case sensitivity（大小写敏感） 211
checking values（检查值） 68
comparing（比较） 435
converting to strings（转换为字符串） 132, 185
declaring（声明） 60, 66, 150
declaring inside loops（循环中声明） 76
declaring inside methods（方法中声明） 116
functionality （功能） 60
hovering over while debugging（调试时停留在上面） 64
loops and（循环） 65
memory considerations（内存考虑） 128
name considerations（命名考虑） 138, 211
objects and（对象） 139
range（范围） 690
renaming（重命名） 757
setting（设置） 68
try/catch blocks and（try/catch块） 489～490
useful data types（有用的数据类型） 61
viewing value changes（查看值改变） 63～64
(参见 reference variables)
View menu（View菜单）
Properties window（Properties窗口） 33
Toolbox option（Toolbox选项） 12
virtual keyword（virtual关键字）238, 244, 248–249
virtual machines（虚拟机） 45
virtual methods（虚方法）
defined（定义） 243
overriding（覆盖） 238, 244, 248～249, 353
Visual Studio Express, 11, 35
Visual Studio IDE (参见IDE)

void (return type)（void（返回类型）） 88～89, 115

# W

Watch window（监视窗口）
adding watch for variables（为变量增加监视） 64
functionality （功能） 476, 481
reproducing problems in（再生其中的问题） 475
viewing results in（查看结果） 353
viewing Unicode values（查看unicode值） 373
WCF (Windows Communication Foundation)（WCF（Windows通信基础库）） 760
where clause（where子句）
functionality （功能） 692, 695～696, 703
LINQ support（LINQ支持） 690
while loops（while循环） 65, 69, 73
whole numbers（整数） 126, 128
Window menu（Window菜单） 9, 11
Windows calculator（Windows计算器）
converting from hex to decimal（十六进制转换为十进制）, 446, 450
Programmer mode（Programmer模式） 127
Scientific mode（Scientific模式） 131, 446
Windows Communication Foundation (WCF)（Windows通信基础库（WCF）） 760
Windows Forms applications（Windows窗体应用） 42, 48
Windows installer（Windows安装程序） 7, 36
Windows Picture Viewer（Windows图片查看器） 616
Windows Presentation Foundation (WPF)（Windows表现基础库（WPF）） 764～765
WindowsApplication1.csproj, 43
WPF (Windows Presentation Foundation)（WPF（Windows表现基础库））764～765
writing data（写数据）
BinaryWriter class（BinaryWriter类） 449
decision-making when（决策） 434
encoded（编码） 449
excuse manager program（借口管理程序） 429～433
File class（File类） 422
FileStream class（FileStream类） 410
serialized files（串行化文件） 451～452
Stream class（Stream类） 408～409
StreamWriter class（StreamWriter类） 411～412, 415
switch statement（switch语句） 436

# X

XML comments（XML注释） 736～737
XML documents（XML文档） 582, 688, 762～763

# Y

yield return statement（yield return 语句） 753～755

# Z

zoo simulator application（动物园模拟系统应用） 228～234, 288
zooming images（放大图像） 629, 631, 633～635